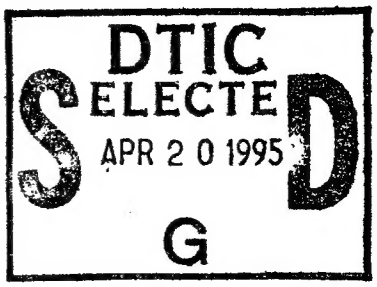
RL-TR- 95-12
Final Technical Report
February 1995

KBSA CONFIGURATION MANAGER

Software Options, Inc.

Michael Karr



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-95-12 has been reviewed and is approved for publication.

APPROVED:

DOUGLAS A. WHITE
Project Engineer

FOR THE COMMANDER:

HENRY J. BUSH
Acting Deputy Director
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

Contents

1 Background

In this project, we designed and prototyped a change and configuration management (CCM) facility for use in the KBSA. This facility was based on the coupling of three independently developed technologies: the *Plexes System*, the *Artifacts System*, and the *Activity Coordination System*.

- The Plexes System is the overall repository for network-wide data; its main role is to provide active views on the data.
- The Artifacts System is the repository of both design objects and derived information; it provides versioning, browsing, updating, and tool integration.
- The Activity Coordination System provides the communication and persistent execution services on which multi-user cooperation is based.

Each of these has a concise formal basis, developed specifically for its domain of application. The elegance and simplicity of the bases lead naturally to the characteristics of openness and extensibility, properties that we exploited in the course of this work, as we will discuss in more detail below.

In conjunction with TRW, we illustrated the applicability of these technologies by constructing a small but representative set of scenarios based on users (both human and automated) and the roles they play in software development processes that require management of change and configurations. The scenarios served as the basis for evaluating functional and scaling properties of technology. This evaluation, based on the use of our prototype implementations of the above systems, had two primary goals: to assess the applicability of the CCM to large-scale software development efforts and to plan the implementation of the CCM facility on other candidate KBSA frameworks for large-scale software.

The results of the project are largely reflected in three of its products: the system itself, a comprehensive paper, and a videotape of its demonstration. Each of these happens to address a slightly different audience for the technology. The software, and its associated users' manual, is available for anyone who wants to use the technology. It is available on half-inch magnetic tape, by request, and can be made available on the Internet, also by request. The paper describes both the theoretical underpinnings of the system and its current implementation. It is written as a submission to a technical journal. The videotape is the TRW report. This may appear to be an unorthodox format, departing as it does from the usual paper report, but it is quite appropriate in this context: it is not possible to convey adequately the dynamic nature of the Activity Coordination System in a static paper medium, even one with pictures. The demonstration provides a non-technical description of the system. We are including, as appendices, copies of the overall design for the CCM facility, a paper describing the Artifacts System, and the technical paper describing the Activity Coordination System. Section 5 contains a complete list of the documents produced by this project. We have not included copies of all of them with this report to avoid any unnecessary reproduction or mailing costs, but we are, of course, happy to provide copies to anyone on request.

2 Summary of Accomplishments

In order to offer superior change and configuration management for the evolution of the considerable amount of KBSA software written in Common Lisp, we extended the Artifacts System to support Common Lisp. This involved the addition of the artifact types `clisp-source` (to hold fragments of source), `clisp-module` (corresponding to compilation units), and `clisp-program` (to specify an overall executable), as well as the derivative classes `clisp-text` (corresponding to the type `clisp-source`), `clisp-file` (input to the compiler), and `clisp-spec` (interface specification). It also involved setting up a "Lisp Worker", the apparatus which allows one to write a deriver in Common Lisp. In spite of apparent major differences between Common Lisp and C, Common Lisp artifacts as developed under this contract and C artifacts developed previously share both a common user view and a considerable amount of implementation. Indeed, part of the work on the contract went into generalizing the underlying machinery to maximize commonality of the implementation. This same machinery is now being used at Harvard University to support a new language for parallel machines.

Part of the work on this contract involved developing a new graphical editor for activity descriptions. This editor has the property of being programmable using a combination of EMACS Lisp and PostScript. It uses an off-board process to interface to X; this process transmits events back to the parent EMACS process. It uses a different off-board process to execute the PostScript; this process communicates low-level graphics commands directly to the X server. Although such an arrangement sounds as if it might be slow, in fact the performance is more than adequate. The ease of extending this editor more than repaid the investment; without it, we would not have had a system that was usable by TRW at a sufficiently early stage in the contract.

We made several significant enhancements to the semantics of the Activity Coordination System. One of these was the implementation of condensation nodes, somewhat analogous to function calls in a procedural programming language, complete with parameterization. The idea is that a user can place a condensation node in one activity description, name another activity description in the condensation node, supply actual parameters, and specify how the "dangling arcs" of the named activity description attach to nodes in the activity description where the condensation node occurs. Further, when viewing a running activity description, the user may ask to view the "subinstantiation" corresponding to the condensation node or may ask to view the instantiation of which a current view is a subinstantiation, thus navigating around the instantiation as a whole. (Needless to say, the programmable graphics editor was a key element in being able to provide this functionality within the given resources.)

Another enhancement was the implementation of multisite activities. A user may indicate that a subgraph of an activity description is to be instantiated at a particular site (given by a constant or a parameter), rather than at the site where the rest of the instantiation occurs. This property of specifying sites provides a truly wide-area functionality for the system: at a demonstration given at Rome Lab, part of an activity description was instantiated locally and part of it on machines at Harvard University (any other Internet site would have worked as well).

One of the improvements we made to the system was that of greatly simplifying the

process of implementing a new primitive activity description (i.e., one that is implemented in C rather than specified graphically). We thus were able to make necessary additions of primitives more efficiently and lowered the barriers for others making such extensions. One of these primitive node types is *view-plex*, discussed more thoroughly in the next section; it provides a connection between the Activity Coordination System and the Artifacts System. (*The Activity Coordination System Extenders' Guide* [Kar] provides full documentation on how to define new primitive node types.)

In addition to the major enhancements to functionality given above and numerous minor ones as well, we markedly improved the performance of certain parts of the system and strengthened the error checking and reporting.

We also invested a significant amount of effort in interacting with our subcontractor, TRW, a subject discussed more fully in the next section.

3 Lessons Learned

The interaction with TRW on this project provided a beneficial contact with The Real World (this is not what TRW thinks their name means). As we expected this exercise was useful in making the Activity Coordination System more robust. Further, it vindicated several essential features of our work. For example, the process that TRW chose to implement with our system was already expressed in their own graphical notation. Naturally, the TRW notation, let us call them "process diagrams", was not identical with that of activity descriptions, the notation used by the Activity Coordination System but, on the whole, the match was reasonably close. Simply the exercise of transcribing process diagrams (which have no execution mechanism and thus no mechanical check on their correctness) into activity descriptions revealed vagueness and other inconsistencies in the process diagrams. Further, the biggest discrepancy between the two systems of notation turned out to point up the importance of the openness which we have touted as a fundamental property of our system. This discrepancy concerned a notation of TRW's process diagrams which meant, loosely, "the database." The idea in a process diagram was that a given agent would, at various points in the process, enter some data into the database," thereby completing a certain subtask. Other subtasks (typically managed by different personnel) would then be triggered by the entry of such data.

When we began work, the Activity Coordination System had no notion of "the database" and, of course, no notion of triggering a subtask by the entry of data. However, the underlying Plexes System, which provides the basis for both the Artifacts System and the Activity Coordination System, already had the notion of a "viewer." Further, the implementation of the "very long execution mechanism," which provides for the persistent execution of activity descriptions, defined the **activations** plex. We were able to implement "the database" and triggering from it with a few simple extensions.

- We changed the notion of viewing to allow an extensible set of viewer types. The only previously existing use of viewers was to update users displays. We set up one type to correspond to the existing host-port (for updating users displays) and another whose purpose was to send plex changes to an activation. The latter, in other words,

"wakes up" the activation in the usual way (creates a Unix process for it) and sends it a message.

- The set of plexes is extensible. To simulate TRW's existing database for the purpose of this study, we added a new plex, the **db** plex (more details below).

- The set of node types of the Activity Coordination System is extensible. We added a new node type called *view-plex*. This node is general enough to view any plex, although in the context of this project, we used it only to view the **db** plex.

The **db** plex is quite simple. It implements a map from a key to a datum. Conceptually, all possible keys are in the plex; however, all but a finite number have the empty string as a datum. The operations on the **db** plex are enter—given k and d, enter $\langle k, d \rangle$ into the plex—and *lookup—given k.* find the current corresponding d. Thus, to remove an element with a given key, enter that key with an empty datum; to test whether a key is in the database, look up the key and test whether the resulting datum is null.

All plexes have "filters", whose purpose is to allow a viewer to see only part of the plex; each plex defines its own notion of filter. For the **db** plex a filter is a regular expression over the key and datum as a whole. The view-plex node allows specification of a filter for the plex it is viewing and, in fact, a bit more: it allows the view-plex node for the **db** plex to be triggered on the appearance, disappearance, or change to an entry.

The *view-plex* node type, although new, is in fact related to the existing node types *user-interface* and *timer*. A node of any of these three types is quiescent most of the time but, when activated, normally goes through the same basic steps.

- It notifies the outside world of its interest in something. (A *timer* node sets a timer running, a *user-interface* node notifies a user that input is to be supplied, and a *view-plex* node makes its activation a viewer on a plex.)

- It waits for something in the outside world to send the activation a message.

- It receives the message and supplies output of the node.

- It re-enters the quiescent state.

The similarity of these node types is apparent to the user, who sees analogous editing characteristics and semantics, and is exploited by the implementation, since the three node types share a significant amount of common code.

In summary, with surprisingly little work, we were able to support a style of activity description that was completely unanticipated at the start of this project and to do so in a way that was compatible both with TRW's existing approach and with the style of the existing Activity Coordination System.

Our experience regarding the use of the Artifacts System at TRW is more complex. Part of the TRW principal investigator's interest in our work originated in the fact that TRW expends a significant amount of resources in "CM" (**configuration management**). There is a CM organization within TRW whose sole purpose is to be trusted with passwords which programmers don't know: to release a module, a programmer sends a message to a member

of the CM organization, who then copies the indicated file(s) to an area that the programmer cannot modify. (Part of the work; with the Activity Coordination System which TRW did was concerned with just this process.) The Artifacts System suggests an entirely different approach to the CM problem, based on the inherent immutability of artifacts. Rather than having to copy files as the basic release action, the status of artifacts can be indicated by attributes. With some additions to the current capability of the Artifacts System, the permission to set attributes can be controlled, for example, by being limited to certain people, or to certain programs (for example, a test program), or even to a particular activity description (for example, to ensure that proper administrative actions have been taken).

The Artifacts System represents a radically new approach to the CM problem, which is now addressed, however inefficiently, by techniques that are ingrained both in individuals and organizations. Add to this the fact that the Artifacts System is presently integrated into only one editor and that, despite its daily use in research organizations, it still lacks some features and performance. After some initial investigation, the task of carrying out even a pilot study of how use of the Artifacts System might affect configuration management at TRW was deemed to be beyond the scope of this project, which is why we focused on use of the Activity Coordination System to assist in the current CM process. We still believe that adoption of the Artifacts System with some extensions could greatly reduce CM costs at TRW, but we need to find a path by which the transition can be managed.

4 Next Steps

As its name implies, the Activity Coordination System has as its goal assisting structured communication within groups of people rather than supporting specific technical activities such as programming or hardware design. Because of this fact, the natural platforms for the Activity Coordination System are personal computers, which serve a wider and less technical user community than Unix workstations. A current goal is to obtain government or commercial support for a port to these other platforms.

As we indicated above. the Artifacts System represents a crucial component of the most promising approach to broader configuration management issues. we have an increased awareness of just how difficult the transition to this new regime will be and are continuing to look for opportunities to explore technology transition paths that would enable widespread adoption of these ideas.

5 Bibliography

[Kar] Michael Karr. *Extenders' Guide for the Activity Coordination System*. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138. Continuously updated.

[Kar93a] Michael Karr. Activity coordination plexes. Working Document, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138. 1993.

[Kar93b] Michael Karr. The activity coordination system. Software Options, Inc., 22 Hilliard Street. Cambridge, MA 02138. September 1993.

[Kar93c] Michael Karr. Activity description editor. Working Document, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, 1993.

[Kar93d] Michael Karr. A programmable graphics editor based on Emacs, the X toolkit, and Ghostscript. Working Document, Software Options, Inc., 1993.

[Kar93e] Michael Karr. Users' guide to the programmable graphics editor. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, July 1993.

[Kar94a] Michael Karr. Boolean algebras and regular expressions. Technical report, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, 1994.

[Kar94b] Michael Karr. Extender's Guide for the Plexes System. Technical report, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, 1994.

[Kar94c] Michael Karr. Permissions. Technical report, Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, 1994.

[KMS93] Michael Karr, Robert T. Morris, and Chris Small. Dynamic types. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, September 1993.

[KS93] Michael Karr and Chris Small. Procedure based nodes. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, July 1993.

[KT] Michael Karr and Judy G. Townley. *Activity Coordination System Users' Manual.* Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138. Continuously updated.

[MKS93] Robert T. Morris, Michael Karr, and Chris Small. Transaction graph implementation. Software Options, Inc., 22 Hilliard Street, Cambridge, MA 02138, October 1993.

[Mor91] Robert T. Morris. Implementation of an activity coordination system. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference.* Rome Laboratory, September 1991.

Appendix A

KBSA Configuration Manager Design

Mike Karr

September 17, 1992

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

This work was supported in part with funds provided by the AFSC, Rome Laboratory, Griffiss AFB, NY 13441-5700 under contract F30602-91-C-0066.

Contents

1 Introduction

Because the foundation of the KBSA Configuration Manager (KBSA CM) is a cross-coupling the Artifacts System and the Activity Coordination System,[1] the bulk of this document addresses the design of that coupling and the required modifications and extensions to each system. We include specifics of the systems only to the extent necessary to understand the proposed work; for further information, refer to [4, 8, 3, 1] for the Artifacts System and [2, 5, 7, 6] for the Activity Coordination System.

Our approach relies heavily on the extensible aspects of the Artifacts and the Activity Coordination Systems. Rather than incorporate knowledge of each directly into the other, we will exploit the fact that each of the systems has "hooks", i.e., places at which functions from outside the system can be attached, with the understanding that these functions are called in specific situations. Thus the coupling of these systems will be based on the fact that each can provide hooks for the other. Except for those specific connections, we will enhance both the Artifacts and the Activity Coordination Systems so that the coupling will be more effective, but this will always be done with the criteria of independence and extensibility foremost, so that each system continues to appear as if its design was uninfluenced by the other's existence.

To understand the software design for the KBSA CM, it is necessary to understand the Plexes System, on top of which both the Artifacts and Activity Coordination Systems are implemented. For present purposes, it is sufficient to think of a plex as a container of data. The most important feature of plexes is not the nature of the contents, but the facts that they may be viewed and that views are active (they are updated as the contents change). For example, there is an `artifacts` plex that holds the set of all artifacts; the Artifacts System also uses several other plexes to hold information, for example, installed tools or which jobs are currently running. Similarly, the Activity Coordination System uses several plexes, e.g., one for the set of all ongoing activities.

In constructing the KBSA CM, we will make modifications to and extensions of the Plexes System as well as the Artifacts and Activity Coordination (sub)Systems that live on top of it. The goals of this work may be described succinctly as:

- monitoring—it must be possible for an activity to monitor changes to plexes.
- control—it must be possible to limit what a user can see and do.

The software that we will build may all be traced to these fundamental requirements.

2 Plexes

2.1 Monitoring

We consider how an activity can monitor changes to plexes. To understand this design, it is first necessary to understand how views on a plex are maintained as the contents change. At

[1] Called "transaction graphs" in the proposal.

a very low level, what happens is that any process that makes a change to a plex is obligated to send "change messages" to all the viewers of the plex. These messages are the means by which views are updated; the size of a message is proportional to the size of the change, not the size of the view. The algorithm to send change messages looks in the `plex-viewers` plex (one of the few fundamental plexes in the system) to find out who is viewing the plex. In the current implementation, "who" is a host, port, and an "internal address", uninterpreted by the Plexes System. The process making the change sends the change message, preceded by the internal address, to the port on the host, using standard network communication. The scheme clearly requires that the viewing process register the view in the `plex-viewers` plex. In doing so, it chooses the internal address, which it necessarily must interpret when a change message arrives.

To make it possible for an activity to monitor changes to plexes, it is natural to tap into the viewing machinery. However, an activity does not have a Unix process associated with it, so it is not sensible for the "who" in this case to be a host, port, and internal address. Hence, a reasonable generalization of the `plex-viewers` plex is that where it now uniformly has a host and port, it will in the KBSA CM have either a host and port or a pointer to the activity, i.e., enough information to send the activity the change message. It is also reasonable to retain the notion of an internal address that is sent along to the activity so that the activity can more precisely decide what to do about the change.

The proposal to generalize the host and port in the `plex-viewers` to include an activity pointer is headed in the right direction, but does not go far enough. It violates the principle that the Plexes System "knows" as little as possible about the plexes within it, other than their existence. Currently, the Plexes System knows about only two plexes. One of these is the `plex-viewers` plex, which it uses to send change messages (as we just discussed); the other is the `plexes` plex, which the user can examine to determine the set of currently installed plexes, and which tells the Plexes System about the currently installed functionality for each plex. In a strong sense, the Plexes System does not know about, for example, the `artifacts` plex—one could delete this plex and the Plexes System could still be used for other purposes, e.g., activity coordination.

We know that the desired effect is that we have either a host and port or an activity pointer in the `plex-viewers` plex, but we do not want to build into the Plexes System any knowledge of activity pointers. The only way out is that the "who" field in the `plex-viewers` is an axis of extensibility of the Plexes System—a new such axis, because presently it is always a host and port. There is a standard way in which the Plexes System provides such extensibility. The essence is that the "who" field will be revised to hold both a "who-type" and a "who-value". Loosely, the type is used to locate a function responsible for sending change messages to this type of destination. The Plexes System, instead of just sending to a host and port, locates the send-function and applies it to the who-value and the concatenation of the internal address and the change message, all without really understanding what the send-function might do.

To recover the current functionality, we define the host-and-port type and accompanying send-function that sends the internal address and change message to the host and port, just as at present. To obtain the functionality for activities, we define a different who-type and its send-function as part of the Activity Coordination System, *not* as part of the Plexes System,

which will continue to be ignorant of the Activity Coordination System, in the sense that its plexes could all be deleted, and the Plexes System could still be used for other purposes, e.g., artifacts.

The decision not to build knowledge of activities directly into the Plexes System thus motivates a more extensible design, in a way that we can precisely characterize: the "who" field of the `plex-viewers` plex will be an axis of extensibility. Thus, if a need arises for a system other than the Activity Coordination System to receive change messages, it will not be necessary to make any further extensions to the Plexes System. All that will be necessary is to provide another extension along this existing axis. The end result is that by extensions along this axis, *any* system can monitor changes to plexes, in much the same way that a person can now monitor changes by looking at an active view.

2.2 Access Control[2]

2.2.1 Introduction

Before embarking on the details of access control for the KBSA CM, we wish to distinguish between access control and security. We are *not* attempting to provide a secure system, in the sense that it resists penetration with malicious intent. First, the system is prototyped on Unix, whose security capabilities are not up to the task. Second, security is a notoriously difficult topic, and working on it would divert resources from tasks more central to the success of the KBSA CM. When we use the term "access control", what we mean is appropriate limitations on the actions of a user who issues only commands that are part of the KBSA CM system. Our goal is to develop a sensible, usable system. We believe that an implementation on a suitable secure platform, perhaps trusted Mach, would add the security component.

Access control in the Plexes System is currently inadequate for the KBSA CM. In this section, we discuss the necessary enhancements. As with our enhancements of the viewing apparatus, the approach will be to build into the system as little as possible. The enhancements that are made will take the form of axes of extensibility, in order to make the system as robust as possible in the face of future requirements.

An important issue in any system of access control is its efficiency. There is an inevitable trade-off between the specificity of access control and its speed. Our design goal is to choose the axes of extensibility that are independent of specificity and speed, i.e., the trade-off is made in the extension rather than in the system. This is a delicate matter, but not directly at odds with the principle of ignorance.

Because the Plexes System is capable of interacting with several different databases, and because access control is such a fundamental part of a database, the question naturally arises as to what could possibly serve as the basis of access control for the Plexes System as a whole. We have chosen public key cryptography as that basis for several reasons. One of the reasons is that the Plexes System relies heavily on communication, and almost by definition, cryptography is a means of securing communication. This is especially important when the application is activity coordination. A related reason is that encryption allows for secure

[2]This section represents some significant changes as well as elaborations of the discussion of the same subject in the first quarterly report.

storage of data in insecure databases with a cost that is related to the degree of security, an important issue if one does not entirely trust the database. Finally, even when security is to be achieved by using the security of the database public key cryptography provides authentication of the user, via digital signatures of identify-friend-or-foe techniques; once identification is made the access controls of the database itself can be used, saving the expense of encryption. In short, public key cryptography provides both the necessary power and flexibility for access control in a context where the database is not a constant.

2.2.2 Permissions

Since we do not wish to build into the Plexes System "knowledge" of a fixed set of operations, we cannot built into any knowledge of access control for its operations. This suggests that we need to set up some extensible scheme by which access control for various operations can be specified.

In common parlance, we speak of "someone" having "permission" to do "something". We have already identified the "something" in this case as an operation, for example, examining the contents of an artifact. The "something" will be characterized more formally as a plex/token pair, where the token is simply a string that names the operation. (In some cases several operations may share the same token.) We have not yet identified the "someone". For want of a better term, we will call the "someone" a *user* , which we will discuss in greater detail below. Finally, for a given plex/token/user triple, there will be a *permission* that limits what a user can do with an operation applied to particular arguments. For example, for the **artifacts** plex, there will be an **examine** token. If I attempt to examine an artifact, the attributes of the artifact in question are tested against my permission for **artifacts/examine**. The result determines whether the function **examine-artifact** will actually show me the contents of the artifact.

A consequence of the approach outlined in the previous paragraph is that a user must become a formal object in the system. In fact, there is already a **users** plex that lists the various users, but this plex is purely an extension--the Plexes System does not know anything about it except that it exists. We will revise the **users** plex so that it associates with each user a be public key and the permissions for various plex/token pairs. We will revise operations that are built into the Plexes System, for example the **view-plex** command (see section 2.2.3 for more details), so that they consult this plex to ensure that user has access to no more functionally than indicated by his permissions. Thus, the **users** plex, with its public key and permissions data, will join the **plexes** and **plex-viewers** plexes as a fundamental plex, i.e. one which the Plexes System consults in performing its own operations. We will also modify the **plexes** plex so that implementors of each plex can define the essential units of permission for that plex.

The detailed design of permissions including required revisions of the **plexes** and **users** plex is documented under a separate title *Permissions*. The *Permissions* document is written from two points of view, the naive user of the system and an extender, specifically someone implementing a plex. These descriptions are sufficiently detailed that they describe a software design for the modifications that we will make to the **plexes** and **users** plexes.

2.2.3 The Connection between Filters and Permissions

A view on a plex does not show everything that is in the plex; rather, the information in a view is limited by a *filter*. For example, a filter on a view of the `artifacts` plex limits the artifacts in the view to those having certain attributes: one view of the `artifacts` plex might have only those artifacts created by "mike", while another view might have artifacts created by "mike" or "doug".

The relevance of filters to the question of access is that they already have the property of limiting what can be seen. A natural way of imposing access control on plexes in general, not merely the `artifacts` plex, is to posit that for every plex p, there is a token `view`, and that a user's permission for any p/`view` is simply a filter for viewing plex p. Thus, each user has, for each plex, what amounts to a "maximal view filter" that limits any view of the plex for that user. More precisely, it would be reasonable to require that filters are ordered under restrictiveness, and that the filter for any view that a user has on a plex is not less restrictive than the user's maximal view filter for that plex.

3 Artifacts

The Artifacts System presently allows only a fixed set of attributes that may be attached to artifacts: type, name, creator, timestamp, and up-to-date. (These are explained in [8].) Further, a filter is a conjunction of conditions on each one of these individually. The main enhancement to the Artifacts System for its role in the KBSA CM is a generalization of this attribute scheme, in three important ways: the set of attributes will be extensible, filters will be closed under boolean combinations, and the notion of up-to-date will be considerably generalized. Because the Activity Coordination System interacts with the Artifacts System via viewing the `artifacts` plex and manipulating permissions for its tokens, enhancements of the generality and specificity of attributes and their filters is crucial to the overall power of the KBSA CM. The following sections discuss each of the three enhancements in greater detail.

3.1 Extensible Set of Attributes

Suppose that it is desirable to add the notion of "releasing" an artifact. To do so, a user would declare a new attribute called "release" and define a release activity that attaches a "release" attribute which is the time of the release. Filters on the `artifacts` plex will then, as one of their conditions, be able to test that an artifact has a release date in some particular time interval. Further, one could specify an activity that monitored newly released artifacts, which would, say, notify interested parties, perhaps in a way that depended on other attributes of the artifact. It is examples like these that support the claim that extensible attributes play an important role in the KBSA CM.

3.2 Boolean Combinations of Conditions

The second generalization of the attribute machinery is that a filter will be an arbitrary boolean combination of conditions on individual attributes. For example, it will be possible to view a set of artifacts that are created by a particular user and released before a given date or that are created by another user and released after another date. As the discussion in the *Permissions* document motivates, permissions must be closed under disjunction, conjunction, and complement, i.e., must be a boolean algebra. Since view filters are permissions, filters for the `artifacts` plex in particular must constitute a boolean algebra, and further, they must constitute an extensible boolean algebra.

Let us consider the task of declaring a new attribute for the `artifacts` plex. It is certainly not reasonable to require that the user, most likely not a mathematician, think in terms of boolean algebras. Instead, the user specifies the domain of values in which the attributes lie. Each of these domains carries with it the set of possible filters for attributes based on that domain. The domains are an axis of extensibility—a deep one because the extender must know about boolean algebra—but this axis is not extended by a user who is declaring a new attribute, which uses an existing domain.

The issues of providing for the extensibility of attribute domains are twofold. First, it is necessary to specify, for the extender, the precise interface that an extension must meet. This interface specification is officially part of this Software Design Document, but because it will become part of the "Extenders' Guides" series (along with [3, 1]), it is separately titled *Attribute Domains*. This document also describes the set of domains that will be done under this contract, and thus initially available. This list of domains will be sufficient for most practical purposes.

Second, it is necessary to provide a large boolean algebra whose elements are the aforementioned "most general boolean combination" of the filters for the individual attributes. Part of the work we have performed on this contract has been to develop the appropriate mathematical construction for this purpose. This work is officially part of this Software Design Document, but is separately titled (*Boolean Algebras and Regular Expressions*) because it is of independent interest and will be submitted for publication.

A significant feature of the new design is that attribute domains will not be specifically tied to any particular plex in the way that they now are to the `artifacts` and `drafts` plexes. Rather, it will become significantly easier to implement a plex that uses attributes, and extensions to attribute domains will then benefit all such plexes.

3.3 Generalized Notion of "up-to-date"

At present, by definition:

- An artifact is up-to-date $\Leftrightarrow$ it has no successors and all of it references are up-to-date.

This notion is inadequate, as demonstrated when there is a branching line of development. Suppose, for example, that there is a released system, and that a developer wants to start a new line of development beginning with that system. With the current uniform notion of up-to-date, the released system is technically out-of-date as soon as the branch is established.

However, there is an important intuitive sense in which the released system is still up-to-date as far as *released* systems are concerned. Suppose further that while the new line of development is being pursued, a small bug is discovered in the released system. Then it is perfectly reasonable to supersede the released system with a system which incorporates the fix to the new system. After doing so, the original released system is intuitively out-of-date even as a released system. (The issue of incorporating the fix in the branching line of development is a different matter, one which the developer must worry about.)

It is important that the KBSA CM be able to capture formally the intuitive notion that the property of being up-to-date is relative, because keeping configurations "up to date", for varying definitions of "up to date", is an important action in change and configuration management. Since the notion of filters is already present in the system, it is natural to relativize with respect to filters, and make the following definition:

- An artifact is up-to-date with respect to a filter $f \Leftrightarrow$:
  - There is a null intersection of the artifacts in the transitive closure of its successors and those in a view filtered by f.
  - All of its references are up-to-date with respect to f.

Thus, the current property of being up-to-date is the new property of being up-to-date with respect to the filter that passes all artifacts. To solve the problem posed in the previous paragraph, one could introduce a boolean attribute "released" (which might also be used in activity coordination). Then the intuitive notion "up-to-date as far as released systems are concerned" would quite formally be "up-to-date with respect to the filter that passes all artifacts whose 'released' attribute is true".

With this generalization, "up-to-date" will no longer be implemented as an attribute. However, the syntax for filters will treat it just like other attributes, so that the user has a uniform way of specifying such filters. The key difference between "up-to-date" and true attributes is that whether an artifact is up-to-date depends crucially on its successors and references—you cannot, for example, "set" the up-to-date-ness of an artifact arbitrarily to true or false.

4 Activity Coordination

Recall that in section 2.1 we described the "hook" by which the Plexes System can announce changes to plexes in quite arbitrary ways: the "who" field of the `plex-viewers` plex will be an axis of extensibility. The subject of this section is the particular extension for the Activity Coordination System. The implementation necessarily reflects the layered implementation of the Activity Coordination System, which is an implementation of transaction graphs on top of a very long execution mechanism. The first two sections below discuss these layers separately.

4.1 Very Long Execution

The very long execution mechanism provides for persistent execution of programs. The fundamental object is an "activation", an analogue of what is called a "process" in the Unix operating system. An activation may be "awake" in which case there is an operating system process that corresponds to it, or it may be "asleep", in which case the state of the process is recorded as an executable file (to be run which it is next awakened) and a "state object" (typically a file) that is supplied as input when the process is awakened. Any process may send a message to an activation, using a function in a subroutine package supplied by the system. If the activation is awake, the message is sent to the operating system process; if the activation is asleep, it is first awakened (and supplied its state object), and then the message is sent to the process. There are also facilities for initiating and aborting activations. Also, the process that runs on behalf of an activation may request to go to sleep, or may indicate normal (as opposed to aborted) termination of the activation.

Just as a Unix process has an id (the "pid"), there is an "activation id". The set of currently executing activations is recorded in the `activations` plex, as a map from activation id to the information about that activation, i.e., whether it is awake, if so, where messages to it should be sent, and if not, its executable file and state object.[3]

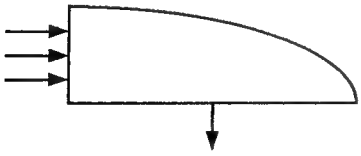
The route by which changes to plexes will be monitored by activities is by a simple extension to the "who" field of the `plex-viewers` plex: this field may be an activation id, in which case, the change message is "sent" to that activation in the above sense, i.e., the activation will be awakened if it is asleep, and in any case, the associated operating system process will receive the message. (Recall the extensibility of the "who" field from section 2.1.)

4.2 Transaction Graphs

A transaction graph ultimately consists of a set of primitive activity descriptions, each of which is implemented by of a set of functions incorporated into executables. Thus, to get information from change messages into a form usable by transaction graphs, it is necessary to make two kinds of functionality available in primitive activity descriptions:

- Modify the `plex-viewers` plex either by adding a new viewer of a plex, specifying a given filter, or change an existing entry to have a new filter, or delete an entry.
- Respond to change messages.

To avoid proliferation of primitive activity descriptions, we will add only one, known as a "view-plex", drawn as a quarter ellipse. All incident arrows use send/receive protocols (see [2] for a discussion of "protocols", including send/receive). The arrows that enter the left edge receive a plex and a filter to add/change/delete viewers, plus a third input whose purpose will be discussed below.



[3]The executable file and state object are retained even when it is awake, for restart in case of failure. This gets into issues that don't directly concern us here.

Because plexes are so general, the Plexes System imposes no special structure on a change message—it is simply a string that all viewers of the plex know how to interpret. This is fine if the viewing process is an editor, because it can be programmed to use the string to make an update to a display and the display is readily understood by a user, even if the details of the change message are cryptic. But it is clearly not reasonable that an activity description, which is supposed to be exceedingly easy to specify, be programmed at the graphical level to parse the change message and present something sensible to the user.

To meet the conflicting goals of avoiding proliferation of primitive activity descriptions and excessive programming at the graphical level, the view-plex activity description takes, in addition to the plex and filter, a *change table*, a plex-specific data structure used to decode the change message and produce or suppress the output of a view-plex node. The idea is that a change table is supposed to be easy to specify by any user who understands the purpose and appearance (but not the implementation) of the particular plex. A running transaction graph will be able to accept a change table as input (see the next section), as will the editor, if one wishes to fix the change table when editing.

To specify the semantics of a view-plex node, we posit the existence of two vectors of functions, each indexed by plex.

- view-plex—given an activation id, node name within the activation, plex, and filter, it establishes the node in the activation as a view on the plex with the filter, and yields a *view structure* (a data structure of the plex implementor's choosing) for that plex.

- review-plex—given a view structure for a plex, a change table, and a change message, it updates the view structure and yields either nil (meaning that the view-plex node is not to respond to this change), or else yields the value to be placed on the output arc of the view-plex node.

The actions of a view-plex node are as follows. With each wave of inputs on the left, a view on the plex is established using the appropriate element of the view-plex vector. The entry in the `plex-viewers` plex specifies that changes to the plex that affect the view cause change messages to arrive at the node. (If the view from this node of this instantiation of the activity description already exists, remove the view (if the filter is nil) or change the filter (otherwise); if the view does not already exist and the filter is not nil, establish a new view). The view structure that results is incorporated into the state of the node.

Change messages will arrive at this node as "out-of-graph" messages (see [2]). When a change message arrives, apply the appropriate element of the review-plex vector to the view structure, change table, and message. If the result is not nil, place the value on the output arc.

This describes an extensible scheme for the viewing of plexes by activity descriptions, but to complete the story, we need to show how a change table will work for the `artifacts` plex in particular, for two reasons: (a) to provide an existence proof that change tables can be made simple to specify and yet provide enough flexibility so that activity descriptions can act on changes in interesting ways, and (b) because the connection between the `artifacts` plex and activity coordination is especially important to the KBSA CM. Change tables for the `artifacts` plex are motivated by two considerations:

- A user of the `artifacts` plex already knows about attributes and filters for it.
- A change to a view on the `artifacts` plex is either because an artifact enters it, leaves it, or remains in it but one or more of its attributes has a changed value.

Accordingly, a change table for the `artifacts` plex is simply a list of the attributes whose corresponding values the view is sensitive to, and the element of the review-plex vector element for `artifacts` behaves as follows.

- If a new artifact enters the view, the result is $\langle \texttt{nil}, p \rangle$, where p is the artifact.
- If an artifact leaves the view, the result is $\langle a, \texttt{nil} \rangle$, where a is the list of attribute/value pairs of the artifact that was in the view (the old artifact itself cannot be in the result if the reason that it left that view was that it was deleted; for uniformity, the result always has only the attributes, not the artifact).
- If an artifact remains in the view but some of its attributes have changed values, the result is $\langle a, p \rangle$, where a is the list of attribute/value pairs of the artifact where the values have been superseded, and p is the artifact (from which current attributes may be obtained directly).

To use the result of a view-plex node in a larger activity description, there are utilities for manipulating artifacts, change tables, attribute/value lists, etc; these are discussed in the next section.

As an example of aiding change and configuration management with an activity description that views the `artifacts` plex, suppose that a project has some notion of a public library, which every one in the project is free to use and perhaps modify; however, it is desirable that any change in the library is announced to all the members of the project. This could be accomplished as follows. Declare a new boolean attribute, say *public-library*, where the value of this attribute is true if the artifact is considered to be in the library by appropriately setting the value of the `public-library` attribute. Then, define an activity description with a view-plex node that views the `artifacts` plex with the filter `public-library & up-to-date (public-library)`,[4] and with a change table specifying only the "name" attribute, so that changes to other attributes are ignored. On receiving a change, the activity description sends a message to all the members of the project. They thus find out when an artifact is superseded by a new version, when artifacts are entered or removed from the library, and when artifacts are renamed.

This is only the simplest of examples; one can imagine much more elaborate connections between low level changes to the status of configurations (i.e., changes in the `artifacts` plex) and high level activity that must precede or follow such changes.

[4] In English, this filter is true of artifacts whose `public-library` attribute is true and which are up-to-date with respect to that property.

4.3 Utilities

The connection between the `artifacts` plex and transaction graphs requires a number of simple utilities for manipulating values related to artifacts in activity descriptions. First, an input node (which is polymorphic anyway) needs to be able to input and output (i.e., display) the following kinds of values:

- artifacts (For example, an activity description might specify the delivery of a design document at a certain point; this would be done by specifying that an artifact is legal input at a particular input node.)
- filters and change tables (Because they are used in view-plex nodes.)
- attribute/value lists (Because they are produced by view-plex nodes.)
- permissions (Because sometimes permissions are granted only after certain activities have taken place.)

It will also be necessary to have procedures for manipulating artifacts and their filters and attributes. By "procedure" here, we mean rather specifically one that can be used in the context of a procedure-based node (see [2]). The following list is minimal, and may be expanded upon:

- artifact-passes-filter—given an artifact and a filter, do its attributes pass the filter?
- get-artifact-attribute—given an artifact and an attribute name, obtain the current value of the attribute of the artifact.
- set-artifact-attribute—given an artifact, attribute name, and value, attempt to set the artifact attribute to the value. Yield true if successful, and false if permission denied.

None of these utilities represent a design effort or much of an implementation effort.

References

[1] Michael Karr. Extender's Guide for the Plexes System. Technical Document, Software Options, Inc. Continuously updated.

[2] Michael Karr. Transaction Graphs: A Sketch Formalism for Activity Coordination. Technical Report RADC-TR-90-347, Rome Air Development Center, December 1990.

[3] Michael Karr, Glenn Holloway, and Steve Rozen. Extender's Guide for Artifacts and Drafts. Technical Document, Software Options, Inc. Continuously updated.

[4] Michael Karr and Glenn H. Holloway. Beyond the Read-Eval Loop: Architecture of the E-L Environment. Technical Report SOI-02-89, Software Options, Inc., August 1989.

[5] Mike Karr and Thomas E. Cheatham. A Solution to the ISPW-6 Software Process. In *Software Process Workshop*, October 1990.

[6] Robert Morris. Activity Description Cookbook. Technical Document, Software Options, Inc. Continuously updated.

[7] Robert Morris. Implementation of an Activity Coordination System. In *6th Annual Knowledge-Based Software Engineering Conference*, pages 264–275, Rome Laboratory, Griffiss AFB, New York, September 1991.

[8] Judy G. Townley. E-L Users' Manual. Technical Document, Software Options, Inc. Continuously updated.

Appendix B

Beyond the Read-Eval Loop: The Artifacts System

Mike Karr

September 2, 1993

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

The purpose of the Artifacts System is to structure complex, evolving data, to assist users in their cooperative effort to develop such data, and to integrate the tools that operate on and produce this data. A key element in the design is to eliminate what is the usual interaction with a computer-based system: run a tool to achieve a desired effect. Rather, users of the Artifacts System set up structures that indicate desired results, and can browse these structures in hypertext-like fashion; tool invocation is usually implicit. Version and configuration management is an integral part of the system, not a facility on the side.

Keywords: configuration management, tool integration, version control

This work was supported in part with funds provided by ARPA, under contracts N00014-85-0710 and N00014-90-0024, and in part by the AFSC, Rome Laboratory, Griffiss AFB, NY 13441-5700 under contract F30602-91-C-0066.

Contents

1 Introduction

The Artifacts System is a product-centered system that addresses areas such as software, electrical, or mechanical design in which:

- Editing is a central activity.
- The data is large and highly structured.
- The data evolves, usually linearly, but occasionally with branches and merges.
- Several people work jointly on a project.
- There is much derived information which is expensive both to compute and to store.
- The set of tools that compute derived information changes (usually grows).
- There is a heterogeneous network of computers which can be exploited as a whole.
- Users interact with the system via bit-mapped graphics.

The challenges and opportunities implicit in these premises require a change in the usual paradigm for interaction with a computer-based system: run a tool to achieve a desired effect. Whether the system is Lisp-style, where the interaction is the successive evaluation of forms, or UNIX-style, with its successive execution of programs, the end result is that the user spends a large amount of time trying to manage a vast sea of relatively unstructured project data. In the Artifacts System, a user sets up a structure that specifies a desired result, and the system is responsible for the details of tool invocation.

2 Fundamental Relations

An *artifact* is the fundamental unit of data in the system. Like a file, an artifact has contents used to store data of interest to users, and like a file system, the Artifacts System does not interpret the contents. An artifact differs from a file in that *its contents never change.* In this respect, it is more like a "version" of a file, as long as one's notion of a file system does not allow changing the contents of a version of a file. The term "artifact" was chosen with its archaeological connotation in mind: small, created some time ago, and unchanging. The immutability property may seem surprising at first, but as we shall see, it plays a key role in providing a proper basis for tracking the evolution of a large and constantly changing structure, such as a program or document.

An artifact has a type, and typing of artifacts plays a central role in integrating tools. The set of types grows as new tools are added to the environment. In classical file systems, naming conventions are often used to achieve something of the effect of types, but there is generally no enforced correspondence between a naming convention and the contents of a file. Types of artifacts are used in a classical programming way: they govern the structure of the contents of an artifact and the operations that can be applied to it. Extenders of the

environment provide new types whose operations conform to prescribed rules (analogous to "method interfaces" or "roles" in object-oriented programming).

Both file systems and the Artifacts System have "fine structure" and "coarse structure" of data. We will use the term "fine structure" to refer to the contents of artifacts. The term "coarse structure" will refer to the constructs that the system provides and uses in locating and grouping the fundamental units of data. In a file system, this is provided by the directory structure, by which we mean the conceptual structure exposed by its interface. For example, the coarse structure of the UNIX file system has soft and hard links, as well as other constructs detectable by UNIX system calls, e.g., inodes. Another property of file systems is that the names by which a user locates files are an integral part of the coarse structure; they typically name branches in the directory, for example. The same names are used by programs that access files. In the Artifacts System, there is a notion of an artifact "pointer", a unique identifier that is used by programs manipulating artifacts, but not by people. These pointers may be involved in an open-ended set of relations or other structures, a topic explored more fully in section 5, where we will see how artifacts are "named" and given other attributes of interest to people.

The following fundamental relations provide the "coarse structure" of the Artifacts System that is of interest to tools:

- references—connects fine and coarse structure.
- derivative—records the results of tool invocation.
- successor/predecessor—tracks the evolution of artifacts.

The following sections discuss these relations in more detail.

2.1 References

A *reference* from one artifact to another occurs at a particular point in the contents of the referencing artifact. For example, the artifact that specifies this document refers, immediately below, to an artifact of type `C-module`:

The `printline` module

Targets

Source

[stdio.h](spec)

global

```
void
printline(s)
  char *s;
{
  printf(s);
  printf("\n");
}
```

What appears in this document is the typeset form of the artifact, but in the interactive use of the system, references appear as highlighted regions on the screen. Commands that take

artifacts as arguments default to an artifact near the cursor, if possible. Since one of these commands is "examine-artifact", the overall effect is that of browsing through hypertext.

The first line of the above inset is the caption. We shall discuss the targets section in connection with the next example. The purpose of the source section is to supply both text to be compiled by a C compiler, corresponding to the usual `.c` file, as well as the interface, or *spec*, that other modules need to use the public functions defined within it, corresponding to the usual `.h` file. The first line of the source section corresponds to `#include <stdio.h>` in an ordinary `.c` file. It is the typeset form of an artifact reference to an artifact of type `universal`; these artifacts are used to link the artifact world to the file world without duplicating information but maintaining the unmodifiability of artifacts—if there is a change to the file to which the artifact refers, the Artifacts System eventually discovers this fact, with very high probability. Thus, the Artifacts System provides a consistent view of immutable artifacts, yet has a pragmatic connection to mutable files.

The user of the Artifacts System has the option of defining the spec (`.h` file) for a `C-module` artifact implicitly. In the above module, the definition of the `printline` function is marked as "global" in the artifact, and the spec for this module is constructed to contain the suitable declaration for `printline`. (The typeset version of the artifact marks such regions with a marginal tag.) The advantage of using this style is that the header of a function appears only once, so it is impossible to have an inconsistent declaration and definition.

The following artifact refers to the spec of the previous one:

The `hello` module

Targets

`sun4` [`gcc` (universal artifact)] `-g`

Source

[The `printline` module]⟨spec⟩

```
  void
main()
{
  [The body of the main program for hello]
}
```

In this example, there is a non-empty targets section. In general, a target section in a `C-module` artifact may have 0 or more lines, each of which specifies a particular compilation of the module. In this case, the one target specifies the machine architecture "sun4" and that the compilation will be done with `gcc`, again using a `universal` artifact to link into the file system; the rest of the line (`-g`) specifies compiler switches.

The body of the function `main` has a reference to an artifact without ⟨spec⟩. This is to a `C-source` artifact, the purpose of which is merely to allow the text that the compiler will see to be broken into small pieces and scattered through a document, for example, here:

The body of the main program for `hello`

```
printline("Hello world!");
```

This particular example is excessively fine-grained, and intended only as an illustration, both

of references and of the way in which the Artifacts System supports the tight integration of programs and documentation. (See the end of section 2.3 for more about LaTeX.)

To complete this example, the following `unix-program` artifact (deliberately uncaptioned) specifies an executable for each of two architectures.

Targets

```
sun4-dbg  [cc (universal artifact)] -g
  C [cc (universal artifact)] -g
mips-pro  [gcc (universal artifact)]
  C [gcc (universal artifact)] -O
```

Modules/Instructions

[The `hello` module]

The targets section of this artifact has two subsections (each beginning with a non-indented line), one for the sun4 architecture and one for the mips. The respective variants (dbg and pro) are simply to name the switch combinations (for the linker) that follow. (Non-blank variants are not necessary here, but would be if there were more than one target for a given architecture.) The dbg variant supplies the `-g` switch, and the pro variant, no switches. On the first line of each subsection is an artifact reference to the linker. In the usual C style, we have used the same program for linking that is used for compiling, but this is not required; for example, one could use `ld` directly.

The indented lines of a target subsection specify how to compile the modules of the program. Each such line begins with a language $\mathcal{L}$ (here, "C") and has an artifact for a compiler, followed by switches. The machinery has much more flexibility than needs to be explained in this paper, and we give only the rules sufficient to explain the above examples.

- If the target section of a module artifact has an entry for the same architecture and variant as that for the executable target, use the object module specified by the module artifact.
- Otherwise, if the module is written in language $\mathcal{L}$ and if there is a line in the subsection of the `unix-program` artifact beginning with $\mathcal{L}$, use the compiler and switches on that line to compile the module.
- Otherwise, issue an error.

Thus, even though the `C-module` artifacts did not say how they should be compiled for the mips, it is still possible to use them in a mips executable. Also, the compilation specified for the `hello` module is not used in the executable on the sun4, because there, the variant is blank, while the `unix-program` artifact wanted a `dbg` variant. (Features not discussed here allow one variant of a program to use differing variants of modules.)

The Modules/Instructions section of the `unix-program` artifact provides the modules that make up the executable, together with instructions (switches) to the linker (for the case in which switches need to be between modules—here, there are no switches). Note that only the `hello` module is given, and not the `printline` module. The default rule is that modules on which other modules depend are included in the list of modules given to the linker. Thus,

if a programmer uses artifact references to indicate dependencies, there is a guarantee that all the relevant modules are put into the executable.

We mentioned earlier that an artifact is somewhat like a version of a file. In this section, we have seen that artifacts have references to other artifacts. This is a bit like a file embedding the names of other files, or rather a version of a file embedding the names and versions of other files. However, there is a crucial difference: unlike a file system (or most configuration management systems, such as RCS [Tic85]), *the Artifacts System knows which artifacts refer to other artifacts* at the coarse level. (The locations of the references within the fine structure of an artifact are known only to corresponding type-specific tools.) This property means that there is a natural formal definition in the Artifacts System for a term that is widely used intuitively:

- A *configuration* is the artifact structure in the transitive closure of the references relation starting at a given artifact (the "root").

The Artifacts System does not allow the deletion of an artifact referenced from another artifact, thereby guaranteeing the integrity of configurations.

When we integrated language tools into the Artifacts System, we chose to represent the compilers explicitly as artifacts. Thus they become literally part of a configuration.

2.2 Derivatives

In section 1, we gave the rule that "a user sets up a structure that specifies a desired result". We can now be precise about exactly what that structure is: a configuration. This section concerns the "desired result", which we shall call a *derivative*. For the rule to hold, it is necessary that a derivative be completely determined by the configuration rooted at the artifact. The rule that a derivative cannot depend on information outside artifacts is why, for example, compilers and switches appear in artifacts—they are responsible for producing a derivative.

An artifact may have several *kinds* of derivatives. For example, the `unix-program` artifact above has derivatives of kinds `executable%sun4-dbg` and `executable%mips-pro`. Similarly, the `hello C-module` artifact has an `object%sun4` derivative. Technically, the *derivative* relation consists of triples $\langle p, k, d \rangle$, where p is the artifact pointer (unique identifier), k is the kind, and d is the derivative; for any given p, k pair, there is at most one such triple. Derivatives are stored as the k-derivative of an artifact. The effect of this in the user's view is that derivatives never rattle around loose in the Artifacts System, in contrast with file systems in which, for example, locating the files that go into a particular executable is by itself an uncertain process, never mind trying to determine what procedure and what options might have been used to derive the `.o` files incorporated into the executable.

The user usually deals directly with artifacts, not their derivatives. For example, the command to execute a derived program takes an artifact as an argument, and looks for a derivative whose kind is of the form `executable%`*machine-variant*, where *machine* (i.e., the architecture) is the same as that on which the command is issued. If there is precisely one *variant* for the machine, then that derivative is used; if more than one, the user is asked to specify the *variant*.

A *deriver* is a tool that takes a single artifact as an argument and produces derivatives. The requirement that a deriver take a single argument is not really a limitation; all it says is that what one ordinarily thinks of as arguments must be gathered into a single artifact. This is necessary to provide the simple model for tracking derivatives described above, and has other advantages which we will see later. Far from being a limitation, the idea that an artifact defines the inputs for a deriver is the basis for the openness of the Artifacts System: it is extended by the joint additions of derivers and types. For example, the purpose of the `unix-program` type is to integrate linkers. Similarly, the purpose of the $\mathcal{L}$`-module` type is to integrate compilers for the language $\mathcal{L}$.

A key element in the design of derivers is that they work by scanning the contents of an artifact and requesting particular kinds of derivatives of references, *regardless of the type of the reference.* For example, a `unix-program` artifact is not restricted to referring to $\mathcal{L}$`-module` artifacts; rather, the requirement is that the module referred to must have a `module-summary` derivative, which in turn says what language the module is written in, which `object%...` derivatives it supplies, and the transitive closure of dependent modules. Thus, the kind says what is wanted from an artifact, and the type says what its structure is; the two together specify a somewhat object-oriented approach to locating a deriver, similar to that found in other tool-integration systems, e.g., [Har87].

The principle that a deriver look only at the derivatives of references is important in the openness of the system because it provides a narrow boundary around the deriver. It means that there is a simple way to write derivers to exploit network-wide computing resources (similar to [LJ87]):

- Phase 1—Scan the contents of the artifacts, and determine which kinds of derivatives are needed from which references. Structure this as a list of pairs $\langle p_i, k_i \rangle_i$ which we call a *request*, and submit it.

- Interphase—Accept the request, figure out which of the derivatives already exist, and if necessary, schedule jobs to compute the rest. If a job to produce the derivative is already scheduled or running, a new job is not scheduled. The jobs are run concurrently as resources become available, and eventually all the derivatives $\langle d_i \rangle_i$ exist.

- Phase 2 (optional)—Using the derivatives obtained so far, submit new requests to gather more derivatives, iterating until sufficient information has been obtained. (This involves an interphase for each additional request.)

- Phase 3—Using the collected derivatives, construct the derivative(s) for this artifact, and inform the system (so that it can record it and give it to jobs that may be waiting).

The difficult part of the implementation is the "interphase" part, but extenders don't have to worry about this—they just provide a serial program.

2.3 Successors and Predecessors

The successor relation is a simple binary relation on artifacts; the predecessor relation is its inverse. They are used to track the evolutionary relationship of artifacts. The number of

successors and predecessors is unconstrained, so this can be used to record branching and merging.

In addition to its obvious role in history keeping, the successor relation also plays a role in detecting simultaneous edits, as follows. An artifact is defined to be *out-of-date* if it has a successor or if it references an out-of-date artifact. (Thus the root of a configuration is up-to-date if and only if all the artifacts in the configuration are up-to-date.) When a user asks to supersede an artifact, i.e., establish a new successor for it, there is a warning if the artifact is out-of-date. The user may decide to proceed—this is how branching is possible—but has at least been warned that a departure from linear evolution has occurred.

From a pragmatic point of view, the most important aspect of the historical relations is their role in incremental tools.

- Even though a derivative is *determined* by its contents and references, it may be *computed* by chasing successor/predecessor links and looking at the contents and derivatives of other artifacts.

For example, suppose we add a comment to the above `C-module` artifact. The `C-module` deriver does not scan an artifact and its predecessor to see if only comments have changed, so the module will be recompiled. However, it does check to see if the result of compilation has produced the same object file, and since the comment does not affect the object file, it will see these are indeed the same. It then discards the new file, and uses the old one as the derivative *even though a timestamp would indicate that it is invalid.* Then, the `unix-program` deriver, which checks to see that either the contents of the `unix-program` artifact have changed (modulo references) or that at least one of the object modules is different, will see no differences. Then without even calling the linker, it knows that the executable will be the same as before, and installs the old executable as the derivative of the new artifact. The advantages of recording historical and derivative relations explicitly, as opposed to the more common naming conventions and timestamps, should be clear—less chance for error and greater opportunities to avoid unneeded rederivation. Further, timestamps are not reliable when compilation is done on a network with clocks that are insufficiently synchronized.

Among derivers so far installed, the one with the most sophisticated incremental behavior is that for LaTeX artifacts. The user sets up a configuration rooted at an artifact of type `latex-root`. The deriver appropriately runs LaTeX, BibTeX, and `makeindex` (as well as whatever typesetting tools have been installed for other artifacts, e.g., for `C-module`, ...) to yield a `latex-directory` derivative, which the user does not directly perceive, but exploits by such commands as `preview`, `ps-preview`, `hardcopy`, and `dvi-file` (to export a dvi file). A `latex-directory` derivative is actually a UNIX directory with the usual files that one associates with LaTeX, plus a few more for bookkeeping. The `latex-directory` deriver for a `latex-root` artifact looks for the same derivative of a predecessor, and for multipart documents, is very sophisticated about the parts of the document on which LaTeX must be re-run, and in some cases can avoid runs that a person cannot because of its manipulation of bookkeeping files. Part of this sophistication stems from a subversion of the usual LaTeX commands for counters and labels which records whether these entities are referenced and/or set by a particular part, so that the deriver knows what parts are affected by a change. Suppose, for example, that a configuration rooted at a `latex-root` is superseded by one in

which the only textual change is to a bibliography entry. The deriver runs BIBTEX; if this causes a change to the appearance of the bibliography, it reruns LaTeX on the part containing the bibliography; if the change to the entry caused a change to a citation, it reruns LaTeX only on sections in which the citation appears. Only those who have used the bibliography mechanism of LaTeX can appreciate the hassle reduction and reliability increase that this provides.

3 Editing

An artifact that is prepared by editing begins its life as a *draft*, which for present purposes is simply an edit buffer whose contents are destined to become the contents of an artifact. Like artifacts, drafts have types. The essential difference between a draft and an artifact is that the contents of a draft change as editing proceeds. A draft may reference (in the technical sense of the word) both artifacts and drafts, but artifacts may not reference drafts—artifacts are stable, and drafts are not. It may in fact reference drafts that others are editing.

Because artifacts are immutable, "editing an artifact" actually creates a new artifact. This naturally happens in stages. First, a new draft is created and its contents are initialized to those of the existing artifact; the draft will have a single predecessor, the artifact. One may also start editing from nothing, in which case the draft is initialized in the canonical way for its type, and the predecessor set is null. Once editing is under way, one may "merge" any number of artifact into the draft, in which case the contents of each artifact are added to those of the draft and each artifact is added to the set of predecessors. (A draft cannot have drafts as predecessors; there seems to be no use for it.)

Regardless of how a draft is started, it is turned into an artifact by *committing* it. This causes the draft to disappear and a new artifact to appear, having the same contents as the draft. If the draft references other drafts, the commit command either first commits the referenced drafts,[1] or balks, depending upon user's preference and whether the referenced draft is the user's own or someone else's (one user cannot commit another's edit). The predecessors of the new artifact are those of the draft. The immutability of artifacts obviates the need for check-in/check-out.

The predecessor relationship for drafts is used not only to obtain the corresponding information for the eventual artifact, it also plays a role in detecting a simultaneous edit. Analogous to the way that we define "out-of-date", we say that an artifact is *almost-out-of-date* if there is a draft which has it as a predecessor or if it references an artifact that is almost-out-of-date. When beginning to edit or merge an artifact, the user is warned if the artifact is almost-out-of-date. Thus, the branch in lineage is caught when starting a draft, not when committing it.

When a new artifact becomes a successor of an old one, not only does the old one become out-of-date, every configuration of which it is a member becomes out-of-date. It would clearly not be satisfactory to require the user to manually edit each of these artifacts to change only the references, and then only to change certain references to their successors. What happens

[1] There is presently no facility for committing artifacts with circular references. This would have to occur in a single atomic transaction, but there are no particular difficulties in doing so.

instead is that starting a draft not only starts the draft the user intended, but also guarantees that certain other drafts exist, automatically creating them, if necessary. The default rule is that a draft will exist for every artifact that is up-to-date and will become almost-out-of-date because of the new draft. (Provisions for non-default behavior are not discussed here.) A draft that is automatically created is called a *system* draft; such a draft does not correspond to any buffer and is determined by its predecessor artifacts, differing only in the references to new drafts. A system draft may be "taken over" for further editing, if desired. When a draft is committed, there is an option to "commit upward", i.e., to automatically commit drafts that reference it, and so on, thus bringing into existence successor artifacts for all the artifacts made out-of-date by the commit. One might worry that propagating the effects of editing would consume tremendous amounts of storage. However, as we will see in section 4, the storage required for two artifacts differing only in references is about what it is for one of them alone. Treating them as two artifacts simplifies bookkeeping of derivatives, and provides a simple mental model for tracking evolution.

An important consequence of the fact that the derivative of an artifact depends only upon the configuration rooted at that artifact is that *derivers may be scheduled automatically.* This is the default when a draft is committed, and may be overridden if desired. Each job is scheduled and then run when its turn comes and when there is a machine on the network capable of doing it. The automatic generation of system drafts, the command to commit upward, and the automatic scheduling of derivers combine to make it easy to create successors of an artifact and all the configurations of which it is part, and to obtain derivatives for all the new artifacts—without ever explicitly invoking a single tool. One simply starts the desired draft, makes the modifications, and commits it "upward".

While we have emphasized the implicit use of derivatives and their automatic generation, it is also true that derivatives may be deleted and rederived under user control. There is thus the possibility that a user may be told that a derivative is possible but doesn't currently exist, for example, when trying to execute a program. In this case, the user has the option of scheduling the job to obtain the derivative, after which the command can be re-issued.

4 Bodies and Handles

In this section, we discuss a simple technique for the solution of two problems. In section 2, we noted that there is a need to connect the fine structure of program pieces (which we have since seen is recorded in artifacts) with the coarse structure necessary to organize the pieces (the references relation). In section 3, we observed the necessity to economically represent two artifacts that differ only because they reference different artifacts.

From the user's point of view, an artifact has contents whose structure is determined by the type of the artifact, and which has occasional references to other artifacts. From the system's point of view, life is more complicated. In section 2 we discussed artifact pointers. An artifact also has a *body*, which is the pointer to where the contents are stored. There is a two-level storage of artifacts, connected by artifact *handles.*

- The contents of a body never contain an artifact pointer. Rather, at each place where the user thinks there is a reference, there is actually an artifact handle. Handles may

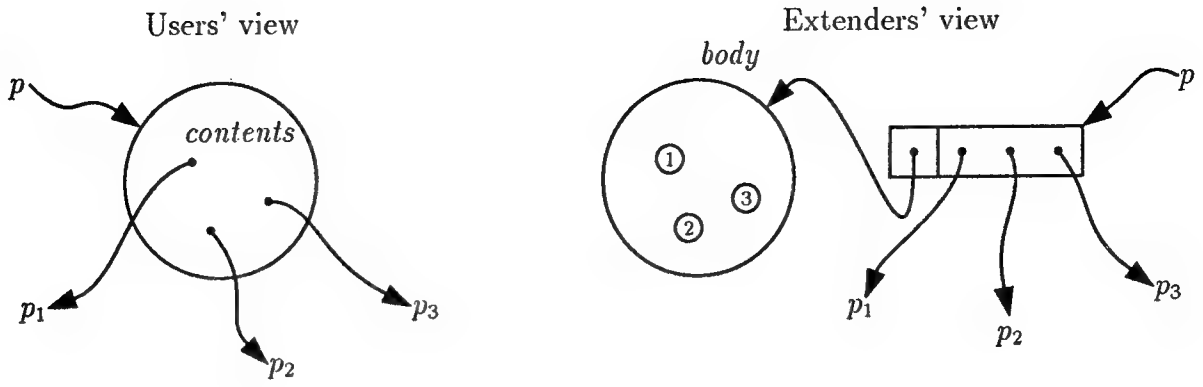


Figure 1: Users' and Extenders' Views of an Artifact p

be thought of as being numbered consecutively: $h_1 \ldots h_k$.

- The system associates with each artifact pointer a pair consisting of a body and a list of artifact pointers (corresponding to the handles in the body).

(See figure 1.) Tools deal with an artifact's body and list of artifact pointers. If they encounter a handle while processing the body, they refer to the pointer list to find the pointer for the referenced artifact. Only type-specific tools know about the structure of bodies, so only such tools can associate handles with artifact pointers. Notice that the system knows nothing about the representation of handles—it never even sees any. A tool is nearly as ignorant of artifact pointers—it uses them only to retrieve a body and pointer list. A serendipitous aspect of recording the list of artifact pointers with each artifact is that this is exactly the references relation discussed earlier.

In addition to addressing the space problem mentioned in section 3, the body/handle technique provides a type-independent means of committing system drafts, using a simple generic "update function": given an artifact A_0 and a pair A_1 and A_1' of artifacts, it produces an artifact A_0' from A_0, where all references to A_1 in A_0 have been replaced by A_1' in A_0'. This function works for artifacts of any type, because it never looks inside the body of the artifact, which is the only place that type-dependent structure exists. Put differently, the update function works at the coarse level.

In the current system, a body is usually a file name relative to a directory specified by the type, but this is only because most of our initial tools have been file-based. With the advent of persistent objectbases integrated with object-oriented languages, we hope that the day will come when derivers can be written with the same mental model that the user has—contents with references to other artifacts. It is not necessary to make an all-or-nothing decision here. Because the body is interpreted on a type-by-type basis, we can have a system in which some types have bodies that are file names, while others are references to objects in POB_1, and still others are references to objects in POB_2, ...

5 Plexes

Up to this point in the discussion, we have been concentrating on artifacts, whose contents are static. It is clear that underlying the implementation of artifacts, there must be some mechanism that allows for objects with dynamic contents. Although the basic ideas in the Artifacts System are independent of the technology used for objects with dynamic contents, the usability of artifacts makes certain demands on the underlying mechanism. This section describes the system that is used in the implementation both on its own terms and in connection with the Artifacts System.

We use the term *plex* to describe an object whose contents are dynamic. The notion generalizes that of "relation" as it is used in database terminology. A relation is a plex whose state at any one time can be described by a set of tuples, all of the same length. Plexes permit the state to take other forms, for example a list or a tree. (It is understood that relations are logically sufficient; the reason for plexes is that sets of tuples may not be natural and convenient for some purposes.) The essence of plexes is not the data structure nor the operations that change state, but rather their ability to be viewed. Secondly, the Plexes System is extensible: a new plex is defined merely by supplying a set of functions having to do with viewing the plex. We will see that the Artifacts System is an extension of the Plexes System.

A user views a plex with an editor, and to the extent allowed by the plex, also uses an editor to change it. Because plexes may be large, viewing is controlled by a *filter*, which limits the amount of the plex that the user sees on the screen. While a filter controls what a user sees, a *format* controls how the information is presented on the screen. The details of what a user sees when viewing a plex through a particular filter, arranged according to a particular format, are of course dependent upon that plex, as are the details of how to specify a filter or a format for the plex. However, the notions of filter and format apply when viewing any plex.

The `artifacts` plex keeps track of all the artifacts in the system and various attributes attached to them. This includes both information that is fundamental—type, body, references, successors/predecessors, derivatives—as well as information that is for the user—name, the person who created it, time of creation (this is the present set; it is subject to extension). For performance reasons, the up-to-date attribute is represented directly, even though it can be computed from the successors and references information.

The `artifacts` plex is the means by which a person locates a particular artifact; for example, users generally organize their data so that there is no more than one element in the set of artifacts that is up-to-date and has a particular name and type, or name and owner. The interface for the Plexes System makes locating artifacts in this way no more difficult than specifying a file name. It is also common to use the viewing machinery to locate artifacts in other ways. For example, a user may wish to see all artifacts with a given type that are up-to-date and created before a given time, or all artifacts with a given name, regardless of whether they are up-to-date or what their type is. Such subsets of artifacts may be viewed by specifying the appropriate filter.

An important feature of the Plexes System is the extent to which displayed information is kept current. If the user is viewing the `artifacts` plex, say through a filter that allows

all users' artifacts provided they are up-to-date, and a format that sorts by time of creation, then if any user commits an edit, the lines displaying the now out-of-date artifacts will vanish, and a line displaying the new up-to-date artifact will appear in the proper place in the display. This will provide the user suffering from file-system withdrawal the illusion of looking at a directory listing that is continuously updated.

When we designed the Plexes System several years ago, we imagined that the idea of maintaining active views on a repository was novel. In fact, there were already instances, and the idea has apparently had multiple independent discoveries: Mercury [Lis88], FSD [WA87], Chiron [TJ93]; there are no doubt others.

6 Present Status and Future Plans

The Artifacts and Plexes Systems were conceived with a graphics-based window system in mind. The prototype implementation uses Epoch, which in turn is built on GNU EMACS and X, which brings us part, but by no means all, of the way toward our ultimate vision for a bit-mapped user interface. The communication mechanism is based on TCP. The core of the system is written in C, which together with Epoch/EMACS, X, and TCP provides a highly portable, freely distributed base; the Artifacts System is public domain.

There are currently around 20 types, for a variety of derivers: LaTeX (used to produce this paper); the C and UNIX types used as examples earlier; Common Lisp and (soon) Ada, in much the same style as C; shellscripts, which allow integration of UNIX files-to-files programs as derivers.

The Artifacts System is in daily use at Software Options, Inc., and has been distributed to several other sites. It has been sufficiently robust to host its own continuing development for several years. Our next project is to couple the Artifacts System with the Activity Coordination System [Kar], so that management issues like bug-report tracking and system release policies have a direct tie-in with the development project activities and data.

7 Conclusion

The Artifacts System provides its users with hypertext-like browsing and editing of complex and evolving objects. These objects may be documents, software, hardware or mechanical designs, ..., or like this paper, may combine objects of different kinds. Version/configuration management is an integral part of the system. Tools are so well integrated that there is minimal explicit tool invocation by the user. The coarse structure of the system provides structural relations that connect an object with its historical relatives and with derivative objects, rather than naming conventions and timestamp comparisons. As well as providing information of direct interest to users, the coarse structure provides a basis for writing highly incremental tools. The system supports the collaborative work of multiple users on a heterogeneous network that is treated as a single multifaceted resource, with the Plexes System providing active views on network-wide project data. While the capabilities of the Artifacts System are both powerful and flexible, it has a simple basis: the reference, derivative, and successor/predecessor relations.

8 Bibliography

[Har87] W. Harrison. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4(6):46–56, 1987.

[Kar] Mike Karr. Technology for activity coordination. Also submitted to this conference.

[Lis88] Barbara Liskov. Communication in the mercury system. In *Proceedings of the 21st Annual Hawaii Conference on System Sciences*, January 1988.

[LJ87] D. B. Leblang and R. P. Chase Jr. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, 1987.

[Tic85] W. F. Tichy. RCS—a system for version control. *Software—Practice and Experience*, 15(7):637–654, 1985.

[TJ93] Richard N. Taylor and Gregory Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In *The First Collected Arcadia Papers*, pages 47–54, 1993.

[Towa] Judy G. Townley. *E-L Users' Manual.* Software Options, Inc., 22 Hilliard Street, Cambridge MA 02138. Continuously updated.

[Towb] Judy G. Townley. *Language Users' Manual.* Software Options, Inc., 22 Hilliard Street, Cambridge MA 02138. Continuously updated.

[WA87] David G. Wile and Dennis G. Allard. Worlds: An organizing structure for object-bases. *SIGPlan Notices*, 22(1), January 1987.

Appendix C

The Activity Coordination System

Mike Karr

September 2, 1993

Software Options, Inc.
22 Hilliard Street
Cambridge, Mass. 02138

This paper describes the Activity Coordination System, a process-centered system for collaborative work. It discusses the underlying mathematical formalism, which was specifically developed for the purpose of describing and tracking distributed, communication-intensive activities. The graph-based semantics encompasses the more familiar Petri nets, but has several novel properties. In particular, it is possible to impose multiple hierarchies on the same graph, so that, for example, the hierarchy with which an activity is described does not have to be the one with which it is viewed.

The paper also discusses the user's (graphical) view of system, by describing a subset of the currently implemented extensions, and the extender's view of the system, by explaining the issues an extender must consider in developing new communication/coordination patterns, which a user can then exploit at the graphical level.

Keywords: software process modeling, process execution, distributed processes, process formalisms

This work was supported in part with funds provided by NOSC, San Diego, CA 92152-5000 under contract N66001-88-C-7008 and in part by the AFSC, Rome Laboratory, Griffiss AFB, NY 13441-5700 under contract F30602-91-C-0066.

Contents

1 Introduction

This paper describes the Activity Coordination System, a process-centered system for collaborative work. It will discuss the underlying mathematical formalism, ways in which this formalism supports various common idioms, the user's and extender's views of the system, current status, and future plans.

The overriding requirement for an activity coordination system is that it be integrated into the everyday activities of the people whose activities are being coordinated. It scarcely needs mentioning that this requires a linguistic component capable of describing a wide range of activities. A further consequence of this requirement is that the system must, at a fundamental level, be *distributed*. Because many organizations are national or international, "distributed" means not only on local area networks (where communication is continuously available), but also on networks where communication is more sporadic (e.g., by occasional dialup). In particular, *the system cannot rely on any centralized execution component.*

A third consequence is that the system must be *extensible*. We assume that the normal activities of its users involve contact with computers, but we cannot assume much more than that—different user communities will use different machines and different software. If a single activity coordination system is to be of widespread use, it must be possible to connect it to a variety of existing software: without such a connection, the use of that software remains unaffected and thus uncoordinated. It is also clearly not desirable for a system to duplicate functionality of existing software. Rather, the system might control access to such software, might supply some of its input, and might look at its output to assist users in subsequent parts of an activity. It is for making such connections that extensibility is a prerequisite—a closed system simply will not be able to supply the necessary degree of integration.

If an activity coordination system is to be successfully integrated into day-to-day activities, it must be *intuitive*, i.e., easily understood by non-technical users. Thus, we are proposing that an activity coordination system must have a linguistic component with which one can express distributed communication and yet is easy to understand and use! We are aware that this is a difficult goal, and it is only by restricting the linguistic component to the very highest level that there is a chance of fulfilling it. While there are many factors involved in making a system intuitive, we feel that a graphical approach is one of the most important. We use the word "graphical" here in two ways: in the mathematical sense, as a structure with nodes and arcs, and in the computer system sense, as a bit-mapped user interface. The use of graphs to describe coordination is quite common: pert charts, organizational charts, and communication networks are examples familiar to people who are not computer experts. Extending the system with fundamentally new coordination paradigms may require programming by experts, but it should be possible for non-experts to connect existing pieces together to provide graphical high-level descriptions of an activity for its participants, who could in turn understand "what is happening" in terms of these graphs, and moreover, could interact with the system via such graphs.

Another requirement for the linguistic component of an activity coordination system is that its constructs be composable and parameterizable. Without these classic concepts from programming languages, it is impossible to have the modularity necessary for the reuse of existing components and the building of large systems.

In addition to the general characteristics of distributedness, extensibility, and intuitiveness, there are several technical issues specific to the arena of activity coordination that pervade the design of a system and thus the underlying formalism:

- user interface—How does a user find out what is going on, and what is to be done next? How is this related to the description of the activity?
- history—How did the project get into its present state? How is the viewing of history related to the user interface (which usually views the present)?
- simulation—How might things go from here?
- cutover—Descriptions of activities change while the activities themselves are under way. How can this be managed?
- summarization—No single person wants (or may be allowed) to see all of the information known to the activity coordination system. How can information be summarized for presentation to the user or archiving as history?

2 Activity Graphs[1]

Before giving the semantics of activity graphs, it is necessary to define their structure. But in understanding the structure, it is helpful to have a glimpse of the key semantic idea:

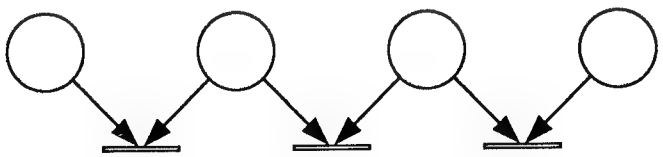
- to use an undirected graph with largely independent computations at the nodes (corresponding to "local" activities),
- coupled with interactions along the arcs (corresponding to the ways in which the activities communicate).

The distributed computation resulting from an executing activity graph is a machine-based microcosm of the real-world activities being coordinated. It guides the users, records their actions, and presents information about the state of the world.

Several important features distinguish activity graphs from message-based schemes like CSP (communicating sequential processes) [Hoa84]. First, the state of each of the computations is visible in a controlled way—the visibility of state provides the basis for presenting information to the users of the system. Second, the use of "interactions" (which will be explained in section 2.2) provides a better basis for activity coordination than mere message passing, because it more succinctly constrains the behavior of the involved computations. Further, limiting the interactions to those that take place in a specified graph guarantees a simple, intuitive means of depicting patterns of communication.

Activity graphs also differ in important ways from Petri nets [Pet], which have also been used as the basis of coordination systems. In fact, the activity graph formalism was motivated by formal deficiencies that Petri nets have for this purpose. First, the semantics of activity graphs are closed under simple graph operations, such as reducing a subgraph to a node or collapsing two arcs that connect the same pair of nodes. The practical consequence of this is that a summary view of an activity looks like, i.e., has the same semantics as, a detailed view. A different way of saying this is that activity graphs allow one to trade off the complexity of graph structure against the complexity of the state and activity (computation) at a node in the graph structure. This is a prerequisite for scaling activity coordination to large organizations, as we will discuss further in section 2.3.

Second, although Petri nets have been used successfully to model many concurrent systems, they are not ideally suited to distributed systems. The problem is the amount of communication required by the firing rule. For example, consider the following fragment of a Petri net, in which the round (state-bearing) nodes are scattered across the world:



How does the system decide which flat (transition) nodes to fire, given various combinations of tokens on the round nodes? It is clear that with concurrently arriving tokens and in

[1] In the original technical report [Kar90], the nomenclature was "transaction graphs", but this term has an entirely different meaning in database theory, and its continued use would invite confusion.

the absence of a central interpreter, the decision to fire a node requires several rounds of communication between it and its inputs. The fundamental interaction in activity graphs requires much less communication (see section 5.2).

More generally, many real-world activities require joint action involving more than two participants, but communication in a distributed system is point-to-point, almost by definition of "distributed". There are many types of multiparty joint action—the Petri net firing rule, contention for a resource, bidding, voting, Byzantine generals, to name a few—and some have interesting, non-trivial implementations. Our approach is to design and build a system in which such actions can be implemented and installed by technically sophisticated "extenders" and exploited by technically unsophisticated "users". Extenders program in an ordinary procedural language such as C, and in fact can implement nodes that provide any of the above joint actions, including Petri nets. Users "program" in a high-level graphical language with node types tailor-made for their application.

2.1 Structure

To describe the structure of an activity graph, we start with the definition of a *signature*, which is written a bit like a procedure header:

- $(n_1 : p_1, \ldots, n_k : p_k)$
  - The n_i are distinct symbols.
  - The p_i are *protocols*, drawn from a set P.

As we will see, such a signature describes a node, and each $n_i : p_i$ is associated with one of its k adjacent arcs. The symbol n_i is the name of the arc relative to the node, and p_i describes behavior on the arc from that node's point of view. We will assume a "transposition" operation which takes a protocol and produces a protocol for the other end of the arc:

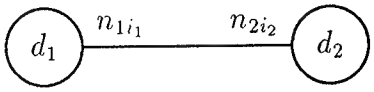
- $^T : P \rightarrow P$, where $p^{TT} = p$, for all $p \in P$.

Perhaps the simplest example of a protocol set is one that directs an arc: $P = \{0, 1\}$, where 0 means an incoming directed arc and 1 means an outgoing directed arc. In this case, $0^T = 1$. Another example of a set of protocols can be constructed from a set of types U, where we let $P \stackrel{\text{def}}{=} U \times U$. This protocol is a pair of types, where $\langle t_i, t_i' \rangle$ means that the node will make visible a value of type t_i, and will see a value of type t_i' made visible by its neighbor along that arc. In this example, we naturally have $\langle t_i, t_i' \rangle^T = \langle t_i', t_i \rangle$. As we will see in section 4.1, protocols may also specify aspects of the behavior along the arc (as the name "protocol" suggests), not just the types of values.

An *activity graph* is an ordinary undirected graph, except that it is allowed to have "dangling arcs" (ones that do not attach to another node). Such a graph is labeled in the following way:

- Each node is labeled with an *activity description*, consisting of several components, one of which is its signature.

- Each non-dangling arc is labeled with two names, one each from the signatures associated with the nodes at its ends. Pictorially, the name appears near the node whose activity description has the signature in which it appears.



  Intuitively, the name indicates how communication on the arc affects the activity described by d_i.

- Each dangling arc is also labeled with two names, one from the signature of the activity description on the node to which the arc is attached (this name appears near the node), and another "graph-relative" name, appearing at the dangling end of the arc.

- At every node, each name in the signature of the activity description appears exactly once as the "near" label of an incident arc.

- A graph-relative name appears on only one (dangling) arc.

- Let n_1 and n_2 be the labels on a non-dangling arc, and let $n_i : p_i$ appear in the respective signatures. Then $p_1 = p_2^T$ (equivalently, $p_1^T = p_2$). This is called the *protocol conformance rule.*

The last rule here is analogous to a rule found in many ordinary programming languages for type conformance between actual and formal parameters of a function. Continuing our example of protocols as pairs of types, if $p_i = \langle t_i, t_i' \rangle$, then the protocol conformance rule says that $t_1 = t_2'$ and $t_1' = t_2$.

We close this section by introducing some notation and a definition that will be used throughout.

- Nodes will usually be denoted by the letter u, and arcs by the letter a; in each instance, subscripts are common.

- If arc a is incident on the node u, the name on a near u is $n_{u,a}$, the corresponding protocol from the signature of the activity description on u is $p_{u,a}$, and the opposite end of a from u will be denoted u/a.

- If a is a dangling arc, then u_a is the (unique) node upon which a is incident, and n_a is the graph-relative name on a.

- The *signature of a graph* is defined to be $(n_a : p_{u_a,a})_a$, where a ranges over all dangling arcs.

2.2 Execution

The first step in defining an "execution" is to describe another component of an activity description d (in addition to its signature):

- An activity description d has an associated set of states $\mathcal{S}_d$.
- In an activity graph, we shall use the convention that $\mathcal{S}_u$ denotes $\mathcal{S}_d$ where d is the label of u.
- The set of states for an activity graph is defined to be $\times_u \mathcal{S}_u$, where u ranges over the nodes of the graph.

An *execution* of an activity graph, defined only when it has no dangling arcs, consists of a sequence of states for the graph, constrained by rules given below. (See section 2.3 for situations in which dangling arcs are allowed.) The rules involve not only states, but elements of the following:

- V, a set of values that are seen along arcs.

There are two remaining components in an activity description, the first of which assigns each element of the signature (intuitively, each arc of a node with this activity description) two functions. The first, denoted σ_i, maps an element of $\mathcal{S}_d$ to an element of V; this is the value that the node shows to its neighbor on that arc.

$$\sigma_{d,i} : \mathcal{S}_d \rightarrow V$$

The second function, denoted τ_i, formalizes what happens during an *interaction*. In essence, a node can change the value it makes visible along *one* arc at a time, based on its own state and its neighbors value on the arc. In conjunction with this operation, it can change its own state. For reasons we discuss below, the state change is non-deterministic, i.e., τ_i maps a state and a value (that exposed by the neighbor) to a *set* of possible subsequent states.

$$\tau_{d,i} : \mathcal{S}_d \times V \rightarrow 2^{\mathcal{S}_d} \text{ where for all } s, v, s' \in \tau_{d,i}(s, v) \text{ and } j \neq i : \sigma_{d,j}(s) = \sigma_{d,j}(s')$$

Intuitively, $\tau_{d,i}$ looks at the node's own state (in $\mathcal{S}_d$) and the value (in V) exposed by the neighbor along the i^{th} arc, and comes up with a new state (an element of $2^{\mathcal{S}_d}$). The condition says that the state change can affect only the value along the arc where the interaction occurs.

The other component of an activity description corresponds to an *out-of-graph* state change, i.e., one not involving an interaction. Out-of-graph state changes correspond either to real world events like the passage of time, or to changes not modeled at a particular level of graph structure.

- Let d have signature $(n_i : p_i)_i$. The component is a function:

$$o_d : \mathcal{S}_d \rightarrow 2^{\mathcal{S}_d} \text{ where for all } s, s' \in o_d(s) \text{ and } i\text{: } \sigma_{d,i}(s) = \sigma_{d,i}(s')$$

The condition says that exposed values do not change without an interaction, i.e., an application of $\tau_{d,i}$ for some i.

Before we can proceed to the definition of execution, there is one last bit of groundwork, concerning how protocols play a role. The idea is that a protocol is a validation predicate on the sequence of values seen along an arc. Since values seen along the arc can appear at

either end, we formalize the sequence as pairs $\langle\langle f_i, v_i\rangle\rangle_i$, where $f_i \in \{0,1\}$; f_i intuitively says which end the value v_i is exposed on. By convention, $f_i = 0$ means that v_i is exposed by σ_i, and thus $f_i = 1$ means that v_i is seen by τ_i. Using S^* to denote the set of all sequences of elements of a set S, we require that:

- An element of P is a predicate on $(\{0,1\} \times V)^*$.

The spirit of activity graphs is that values are *seen* along arcs, not necessarily transmitted, and in this spirit, an element of P is allowed to pass judgment only on the sequence of *changes* to exposed values. This can be technically stated as follows:

- Let $\langle\langle f_i, v_i\rangle\rangle_i \in (\{0,1\} \times V)^*$, and suppose there are some i_1 and i_2 with $f_i \neq f_{i_1}$ for $i_1 < i \leq i_2$, i.e., there is no change in the value seen at the f_{i_1} end of the arc. Then repeating f_{i_1}, v_{i_1} after changes at the f_{i_2} end of the arc does not affect the value of the protocol, or technically, every $p \in P$ must satisfy:

  $$p(\langle\langle f_i, v_i\rangle\rangle_i) \Leftrightarrow p(\langle\langle f_i, v_i\rangle\rangle_i \frown \langle\langle f_{i_1}, v_{i_1}\rangle\rangle)$$

  In practice, we define the action of a protocol as a predicate only for *reduced* sequences, by which we mean sequences with the property that if i_1 and i_2 are successive indices with the same f_i, then $v_{i_1} \neq v_{i_2}$.

The action of $p \in P$ as a predicate must have a proper relationship with the transposition operator on P:

- For all $p \in P$ and $\langle\langle f_i, v_i\rangle\rangle_i$: $p(\langle\langle f_i, v_i\rangle\rangle_i) \Leftrightarrow p^T(\langle\langle 1 - f_i, v_i\rangle\rangle_i)$

In our running example of a protocol as a pair of types $\langle t_0, t_1\rangle$, we would define $\langle t_0, t_1\rangle$ to be true of a sequence $\langle\langle f_i, v_i\rangle\rangle_i$ if v_i has type t_{f_i}, for all i. The point of saying that an element of P is a predicate on a sequence of seen values is that it can thus be used to specify the communication pattern that must be obeyed along the arc, not merely the static properties of individual values.

We shall use the convention that $\tau_{u,a}$ denotes $\tau_{d,i}$, where d is the label of node u, and a is the i^{th} arc incident upon u. Similarly for $\sigma_{u,a}$.

- Given a state of an activity graph, a *subsequent state* is one which differs at exactly one node u, where the new state at u is an element of:

  $$o_u(s_u) \cup \bigcup_a \tau_{u,a}(s_u, \sigma_{u/a,a}(s_{u/a}))$$

  Here, s_u and $s_{u/a}$ are states of nodes from the given graph state, and a ranges over arcs incident upon u.

Finally, we can state the precise definition of *execution*: it is a sequence of graph states, in which:

- Each state but the first is subsequent to the previous element in the sequence.

- On any arc, the values exposed along that arc satisfy the protocol on the arc.

Note that this definition implies that changes at nodes occur serializably—execution does not allow the two nodes on an arc to change simultaneously, i.e., two new exposed values cannot be based on two old exposed values.

We promised earlier some justification for the role of non-determinism. One reason is that activity graphs are designed to help cope with reality, which is non-deterministic. We will also see various technical conveniences of this non-determinism, such as providing an answer to the following question: how do we know when execution of an activity graph is finished? After all, the computational model here is quite distributed (as promised): there is no obvious "exit node", and specifying one, and its behavior, would be artificial, beneath the level of the rest of the above formulation. Rather, the fact that τ and o specify a *set* of states allows for a natural, *distributed*, termination condition.

$$\emptyset = \bigcup_u \Big(o_u(s_u) \cup \bigcup_a \tau_{u,a}(s_u, \sigma_{u/a,a}(s_{u/a})) \Big)$$

Obviously, an execution sequence stops once this condition holds. We shall see other technical conveniences of non-determinism in later sections.

We will address implementation concerns in section 5, but we note here that an implementation of τ does not actually produce a set of states; rather, it eventually picks one state that is a member of the set produced by the theoretical τ. Thus, the definition of τ serves as a specification for the implementation rather than a prescription. Similarly, we do not propose literally implementing the predicate corresponding to an element of P. This too serves as a specification.

In the previous section, we saw that an activity description had a signature, and that there was a transposition operator on the protocols in signatures. These were necessary to specify well-formedness of activity graphs. In this section, we added the following semantic notions, necessary in the definition of execution.

- For activity description d, whose signature is $(n_i : p_i)_i$, there are also the following components.
  - A set of states $\mathcal{S}_d$.
  - A list of map pairs $\langle \sigma_{d,i}, \tau_{d,i} \rangle$, where $\sigma_{d,i} : \mathcal{S}_d \to V$ reveals part of a state to a neighbor and $\tau_{d,i} : \mathcal{S}_d \times V \to 2^{\mathcal{S}_d}$ corresponds to interactions along an arc.
  - A map $o_d : \mathcal{S}_d \to 2^{\mathcal{S}_d}$, corresponding to out-of-graph state changes.
- Each element of P acts as a predicate on values seen along an arc. The action as a predicate is well-behaved with respect to transposition on P.

We close this section by emphasizing the distributedness of the computational model, i.e., its suitability to a truly distributed implementation. Rather than presuming an external scheduler, two adjacent nodes racing to perform an interaction on their common arc can decide between themselves who goes first. It may seem odd that when concurrency is the goal, interactions around a node must occur one at a time, but this is only the way that

one says formally that interactions are serializ*able*, not that the implementation is required to perform them serially. Moreover, this approach to the formalism has the advantage of minimizing the machinery built into the system, and maximizing the flexibility one has in making extensions, i.e., implementing functions specified by σ, τ, and o. This extends even to concepts that are often built into the semantics of a distributed system, like fairness in scheduling.

2.3 Graphical Condensation

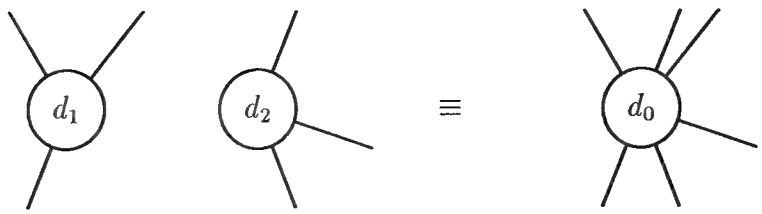
In this section, we will show that there is a natural way to define an activity description that mimics the behavior of an activity graph. This technique of shifting complexity between the graph structure and the activity descriptions has several important consequences. Most obviously, it means that activity graphs can be specified hierarchically—one can place a node in a graph that refers to an entire subgraph, much in the same manner that one writes a subroutine and refers to it by name. The dangling arcs of the graph attach to the arcs that previously were attached to the node.

Less obviously, graphical condensation is related to obtaining different views of an activity, in the following way. Given a subgraph of a graph, that subgraph may be collapsed to a node as a way of summarization: a particular view of an activity may involve a particular condensation of certain aspects of the graph structure. A second view may condense a different subgraph, perhaps partially overlapping the first view. Thus the same activity may be viewed as occurring on quite different, not even hierarchically related, graphs. Both of these aspects make it possible to manage large activity descriptions; the entire graph never has to be seen fully expanded.

To be precise, given an activity graph G, we will define the *induced activity description* d_G from the components of the activity descriptions on the nodes of G and, of course, from the connectivity of G. The dangling arcs of G will be in 1-1 correspondence with the elements of the signature of d_G. Suppose we form a graph G' with no arcs and a single node labeled with the activity description d_G. The result we are going to prove is that G and G' are *semantically equivalent*, by which we mean their executions are in 1-1 correspondence. The proof involves showing that in each of two simple graph reductions—pinching two nodes together and shrinking a loop (an arc with both ends touching the same node)—it is possible to produce an activity description for the new node in terms of the activity description of the one or two (respectively) old nodes of the original graph. It is these technical results, both the result and its proof, that back up the claim that it is possible to shift complexity between graph structure and activity descriptions. They will also motivate some of the technical details of the semantics of activity graphs.

2.3.1 Pinching Two Nodes

In this subsection we consider the induced activity description for the graphical operation of "pinching" two nodes, i.e., replacing them with a single node, and attaching arcs that used to be incident upon either node to the new node. In its simplest form, the graph transformation is:



On the left is an activity graph consisting of two unconnected nodes; on the right is an activity graph consisting of a single node, having arcs corresponding to the arcs on the left, i.e., labeled in the same way (with graph-relative and node-relative names).

We first consider the signature for the new activity description d_0. Let $(n_{ji_j} : p_{ji_j})_{i_j}$, be the signature for d_j. We will assume that the sets of names are distinct, i.e., $n_{1i_1} \neq n_{2i_2}$, for any i_1 and i_2; if not, they may be renamed. Since we are trying to arrange that both graphs appear the same on the outside, they at least have to have the same signature, which forces the signature of d_0 to be defined as:

$$(n_{11} : p_{11}, \ldots, n_{1k_1} : p_{1k_1}, n_{21} : p_{21}, \ldots, n_{2k_2} : p_{2k_2})$$

Next, consider the set of states. If the two graphs behave the same, then in general we would have to have to define $\mathcal{S}_{d_0}$ to be $\mathcal{S}_{d_1} \times \mathcal{S}_{d_2}$. Similarly, identical behavior requires:

- $\sigma_{d_0,i}(\langle s_1, s_2 \rangle) \stackrel{\text{def}}{=} \begin{cases} \sigma_{d_1,i}(s_1) & \text{if } i \leq k_1 \\ \sigma_{d_2,i-k_1}(s_2) & \text{otherwise} \end{cases}$
- $\tau_{d_0,i}(\langle s_1, s_2 \rangle, v) \stackrel{\text{def}}{=} \begin{cases} \tau_{d_1,i}(s_1, v) \times \{s_2\} & \text{if } i \leq k_1 \\ \{s_1\} \times \tau_{d_2,i-k_1}(s_2, v) & \text{otherwise} \end{cases}$
- $o_{d_0}(\langle s_1, s_2 \rangle) \stackrel{\text{def}}{=} o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2)$.

This effectively defines the desired activity description d_0, which we will call $d_1 \times d_2$.

Result 1 Suppose we have two nodes u_1 and u_2 of an activity graph, with activity descriptions d_i, and that we obtain a new activity graph by pinching them into a single node u_0 which we label with $d_1 \times d_2$. Then the two graphs are semantically equivalent.

Proof The "semantic equivalence" can be made precise only by first being precise about the correspondence between states. In the two graphs, the states are given by the following respective sets:

$$\bigtimes_u \mathcal{S}_u \text{ and } (\bigtimes_{u \neq u_1, u_2} \mathcal{S}_u) \times \mathcal{S}_{u_0}$$

But $\mathcal{S}_{u_0} = \mathcal{S}_{u_1} \times \mathcal{S}_{u_2}$, so there is a clear correspondence between states, which involves only re-ordering and restructuring tuples.

$$\langle \ldots s_1 \ldots s_2 \ldots \rangle \leftrightarrow \langle \ldots \quad \ldots \quad \ldots \langle s_1, s_2 \rangle \rangle$$

With this correspondence of states, it is clear from the definition of σ_{u_0} and the fact that σ_{u_i} is unchanged for $i \neq 1$ or 2, that at corresponding states, the same values are exposed at the corresponding ends of all arcs.

The key to the argument is to look at the possible subsequent states of corresponding states. For any subsequent map in the "before" graph which differs at a node other than u_1 or u_2, there is trivially a subsequent map in the "after" graph differing at a node other than u_0, and vice versa. If the change occurs at u_i, changing s_i to s_i', there will be a subsequent map for the "after" graph in which $\langle s_1, s_2 \rangle$ is changed to $\langle s_1', s_2 \rangle$ or $\langle s_1, s_2' \rangle$. Conversely, any change at u_0 will be of this form, and thus there will be a corresponding subsequent state in the "before" graph. In short, there is a 1-1 correspondence between subsequent states, and hence a 1-1 correspondence between executions.
□
Define two activity descriptions to be equivalent, denoted $\cong$, if they have the same signature, if there is a 1-1 correspondence between states, and if under this correspondence, τ, σ and o are all equivalent.

Result 2 $d_1 \times d_2 \cong d_2 \times d_1$ and $(d_1 \times d_2) \times d_3 \cong d_1 \times (d_2 \times d_3)$

Proof In the first case, the correspondence between states is that between $\mathcal{S}_{d_1} \times \mathcal{S}_{d_2}$ and $\mathcal{S}_{d_2} \times \mathcal{S}_{d_1}$, the details of the proof are not worth writing. In the second, we use associativity of cartesian products to get correspondence of states, and give the proof for o.

$$\begin{aligned}
& o_{(d_1 \times d_2) \times d_3}(\langle \langle s_1, s_2 \rangle, s_3 \rangle) \\
= \; & o_{d_1 \times d_2}(\langle s_1, s_2 \rangle) \times \{s_3\} \cup \{\langle s_1, s_2 \rangle\} \times o_{d_3}(s_3) \\
= \; & (o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2)) \times \{s_3\} \cup \{\langle s_1, s_2 \rangle\} \times o_{d_3}(s_3) \\
\cong \; & o_{d_1}(s_1) \times \{s_2\} \times \{s_3\} \cup \{s_1\} \times o_{d_2}(s_2) \times \{s_3\} \cup \{s_1\} \times \{s_2\} \times o_{d_3}(s_3)
\end{aligned}$$

Starting from $o_{d_1 \times (d_2 \times d_3)}$, we get the same expression. Details for σ and τ are omitted.
□
To summarize this section, we have shown that given any activity graph G, we can construct a graph semantically equivalent to G with a single node. The activity description for this node is given by:

$$\bigtimes_{u \in G} d_u, \text{ where } d_u \text{ denotes the activity description that } G \text{ assigns to } u$$

By the preceding result, the order in this product doesn't matter.

2.3.2 Shrinking Loops

While the techniques of the previous section condense an arbitrary activity graph to one having only a single node, the activity description for that node is not what we want to call "the" activity description for the graph. Because pinching nodes does not affect the set of arcs, there will be one loop on the single node for every non-dangling arc in the original graph. In this section, we consider the graph transformation of removing a loop.

The reason for the terminology "shrinking" is that while from the standpoint of only the graph structure, it looks as if the loop is simply being removed, from the point of view of the activity descriptions, the semantics for the loop is being pulled inside the node, i.e., turned into an out-of-graph change of state.

Let d have signature $(n_i : p_i)_i$. For convenience, we will assume that the loop has node-relative names n_1 and n_2, so that the signature for d' is $(n_i : p_i)_{i>2}$. We naturally define $\mathcal{S}_{d'}$ to be $\mathcal{S}_d$, and carry over the definitions of $\sigma_{d,i}$ and $\tau_{d,i}$ for $i > 2$. The only non-trivial part of the construction is that what were previously interactions along the loop must become out-of-graph changes, as seen in:

- $o_{d'}(s) \stackrel{\text{def}}{=} o_d(s) \cup \tau_{d,1}(s, \sigma_{d,2}(s)) \cup \tau_{d,2}(s, \sigma_{s,1}(s))$

This completes the definition of d', which we denote by $d - [1, 2]$; more generally we give the pair of indices removed. We will also use the notation $d - a$ when it is understood that the indices arise from an arc a that is a loop at a node labeled by d.

Result 3 Suppose a node u of an activity graph is labeled by d and has a loop a, and that we obtain a new activity graph by removing arc a at u and replacing the label with $d - a$. Then the two graphs are semantically equivalent.

Proof Unlike the case in the previous subsection, here the sets of states are identical. For subsequent states that differ at a node other than u, the correspondence is immediate, as is the case of u when the state change is due to an interaction along an arc other then a. The remaining state changes at u in the "before" graph are either out-of-graph changes, or interactions along a, all of which correspond to out-of-graph changes at the corresponding node of the "after" graph, and conversely. Thus there is a 1-1 correspondence in subsequent states.

□

Not only does the order of shrinking arcs not matter, but the operation "commutes" with pinching:

Result 4 Let i_1, i_2, i_3 and i_4 be distinct indices of the signature of an activity description d. Then:

$$(d - [i_1, i_2]) - [i_3, i_4] \cong (d - [i_3, i_4]) - [i_1, i_2]$$

Result 5 Let i_1, i_2 be distinct indices of d_1, and let d_2 be an activity description. Then:

$$(d_1 - [i_1, i_2]) \times d_2 \cong (d_1 \times d_2) - [i_1, i_2]$$

(The notation here assumes that signature indices of d_1 correspond to indices for the "d_1 part" of $d_1 \times d_2$.)

Proof In both cases, the signatures are the same, as are the set of states and the actions of σ and τ. In the first result, the expression for o for the left hand activity description expands out to:

$$\Big(o_d(s) \cup \tau_{d,i_1}(s, \sigma_{d,i_2}(s)) \cup \tau_{d,i_2}(s, \sigma_{d,i_1}(s))\Big) \cup \tau_{d,i_3}(s, \sigma_{d,i_4}(s)) \cup \tau_{d,i_4}(s, \sigma_{d,i_3}(s))$$

The right hand activity description expands out to a similar expression with i_1 and i_2 exchanged with i_3 and i_4 respectively. So the result follows by commutativity and associativity of "$\cup$".

In the second result, the computation is messier, but for completeness, here it is:

$$\begin{aligned}
& o_{(d_1 - [i_1,i_2]) \times d_2}(\langle s_1, s_2 \rangle) \\
= \; & o_{d_1 - [i_1,i_2]}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \\
= \; & \Big(o_{d_1}(s_1) \cup \tau_{d_1,i_1}(s_1, \sigma_{d_1,i_2}(s_1)) \cup \tau_{d_1,i_2}(s_1, \sigma_{d_1,i_1}(s_1))\Big) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \\
= \; & o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) \cup \tau_{d_1,i_1}(s_1, \sigma_{d_1,i_2}(s_1)) \times \{s_2\} \cup \tau_{d_1,i_2}(s_1, \sigma_{d_1,i_1}(s_1)) \times \{s_2\} \\
\cong \; & o_{d_1 \times d_2}(\langle s_1, s_2 \rangle) \cup \tau_{d_1 \times d_2,i_1}(\langle s_1, s_2 \rangle, \sigma_{d_1 \times d_2,i_2}(\langle s_1, s_2 \rangle)) \cup \tau_{d_1 \times d_2,i_2}(\langle s_1, s_2 \rangle, \sigma_{d_1 \times d_2,i_1}(\langle s_1, s_2 \rangle)) \\
= \; & o_{d_1 \times d_2 - [i_1,i_2]}(\langle s_1, s_2 \rangle)
\end{aligned}$$

In the "$\cong$" step, we are using the fact that since i_1 and i_2 are indices for d_1, $\tau_{d_1,i}(s_1, v) \times \{s_2\} \cong \tau_{d_1 \times d_2,i}(\langle s_1, s_2 \rangle, v)$, for $i = i_1$ and i_2, and similarly for $\sigma_{d_1,i}$.

□

From the first of these results, we can use the notation $d - \{[i_j, i'_j]\}_j$ to mean $d - [i_1, i'_1] - [i_2, i'_2] - \ldots$, because the order doesn't matter. Similarly, if $\{a_i\}_i$ is a set of arcs, we can use the notation $d - \{a_i\}_i$ without ambiguity.

With these results, it is possible to condense an arbitrary activity graph G to a graph with a single node and only dangling arcs, and having essentially the same execution as G. The activity description on the single node, which we call the *activity description induced by* G, is given by:

$$d_G \stackrel{\text{def}}{=} \bigtimes_{u \in G} d_u - \{a \in G \mid a \text{ is non-dangling}\}$$

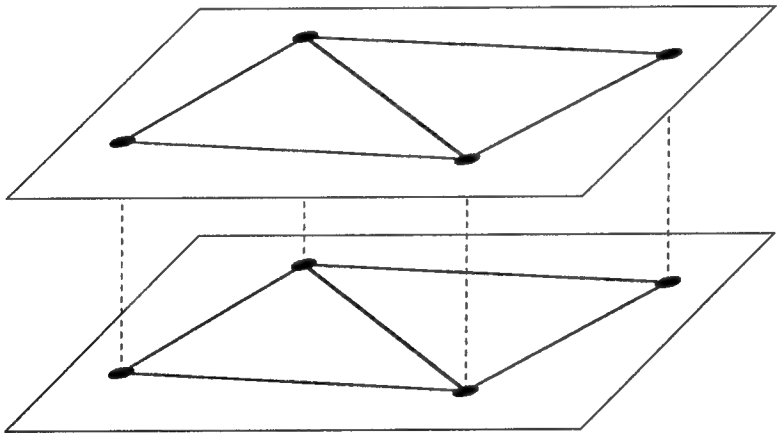
In many respects, *the ability to obtain an activity description that corresponds to a graph dictated the technical details of the definition of an activity description.* For example, suppose that we had defined an activity description to allow simultaneous change of exposed values. Then there would be a class of activity descriptions which could not be realized as activity graphs. Or suppose that we had not allowed out-of-graph changes to the state. Then activity graphs would not be closed under the shrinking of loops.

In summary of these first two subsections, we have seen that the definition of activity descriptions and activity graphs is done in such a way that the complexity of an activity graph can be traded off against the complexity of the state of its nodes. There is nothing about the behavior of an activity description that is particularly different from the behavior of an activity graph; in other words, there is no particular aspect of behavior that *must* be put into a graph, or must *not* be put into a graph. This property enables the construction of a layered system of activity descriptions with clean encapsulation.

2.4 Projected Execution

One often wants to view only part of the information that is necessary for the execution of the activity graph as a whole. The goal of this section is to structure the selection of information in such a way that a view makes sense on its own. The technical approach is to require that the state changes seen in the view are actually an execution in an activity graph that is related in some well-understood way to the activity graph being viewed. Thus, where in the previous section we studied transformations with 1-1 correspondence in executions, here we are interested in transformations which have the property that for every execution in the original activity graph (the one being viewed), there is an execution in the transformed activity graph (the view), but not necessarily vice-versa. That is, the set of executions in the view contains (and may greatly exceed) those in the original graph. This is because of the loss of information in the transformed graph, which causes an increase in the non-determinacy of σ, τ and o.

The techniques of the previous section changed the graph structure, but maintained semantic equivalence. Here, they keep the graph structure the same, and change only the activity descriptions on nodes. Imagine two copies of the same graph structure (not activity graphs, yet), one drawn in a plane directly above the other.



Turn each of these graphs into an activity graph by assigning an activity description to each node. Further, assign a state to each node, where the states on a plane make up the state of an execution going on in that plane. The bottom plane we will view as the detailed execution, and the top plane as a view on the detailed execution—a "filter" through which we look at the "real" execution. This view is formalized as a projection mapping the bottom to the top plane, where this projection behaves "properly" (one might say homomorphically) with respect to the functions σ, τ, and o.

The essence of the subject here is maps from one activity description to another. We will denote such maps with Π, and by abuse of notation, also use Π for the map of constituent parts of an activity description. A map Π must have the following constituents:

- A map that takes signatures to signatures: $(n_i : p_i)_i \mapsto (n_i : \Pi_i(p_i))_i$
- A map that takes a state to a state: $\Pi : \mathcal{S}_d \rightarrow \mathcal{S}_{\Pi(d)}$

- Maps for values seen along an arc: $v \mapsto \Pi_i(v)$
- Maps (functionals) taking $\sigma_{d,i} \mapsto \sigma_{\Pi(d),i}$, $\tau_{d,i} \mapsto \tau_{\Pi(d),i}$, and $o_d \mapsto o_{\Pi(d)}$

We are interested only in the subset of such maps, for which we use the term *projections*, that have the properties given below. In stating these properties, we further abuse Π by applying it to subsets of $\mathcal{S}_d$, by which we mean the subset of $\mathcal{S}_{\Pi(d)}$ obtained by applying Π to elements of the given subset.

- For all $s \in \mathcal{S}_d$, $\Pi_i(\sigma_{d,i}(s)) = \sigma_{\Pi(d),i}(\Pi(s))$
- For all $s \in \mathcal{S}_d, v \in V$: $\Pi(\tau_{d,i}(s,v)) \subseteq \tau_{\Pi(d),i}(\Pi(s), \Pi_i(v))$
- For all $s \in \mathcal{S}_d$: $\Pi(o_d(v)) \subseteq o_{\Pi(d)}(\Pi(s))$

The "$\subseteq$" in each of the last two conditions makes precise the ways in which $\Pi(d)$ may lose information present in d.

In addition to homomorphic behavior at each node, we need a similar condition for what happens on an arc:

- Let Π be a projection of signatures, and let Π_0 and Π_1 be projections of values. We say that Π_0 and Π_1 are *consistent with* Π *at* $p \stackrel{\text{def}}{\Leftrightarrow}$ for any sequence of values $\langle\langle f_j, v_j\rangle\rangle_j$:

$$p(\langle\langle f_j, v_j\rangle\rangle_j) \Rightarrow \Pi(p)(\langle\langle f_j, \Pi_{f_j}(v_j)\rangle\rangle_j)$$

  In other words, Π_0 and Π_1 project a legal sequence of values to a legal sequence in the view.

This property is used in the definition of a projection of an activity graph G, which is a set of projections Π_u, one for each node of G, with the properties:

- Replacing the label d_u on node u with $\Pi_u(d_u)$, for all $u \in G$, results in an activity graph, i.e., one in which the rule for signatures is met. (In other words, if u_1 and u_2 are connected by arc a, and if u_i has protocol p_i on a (so $p_1^T = p_2$), then $\Pi_{u_1,a}(p_1^T) = \Pi_{u_2,a}(p_2)$.)
- For every arc, the projections of the values at the ends of the arc are consistent with the projection of the protocol on the arc.

Again overloading Π, let $\Pi(G)$ be a projection of an activity graph G, and for any state $\{s_u\}_{u \in G}$ for G:

- Let $\Pi(\{s_u\}_{u \in G}) \stackrel{\text{def}}{=} \{\Pi_u(s_u)\}_{u \in G}$

Finally, we need to define what it means to project an execution of a graph. A naive definition would be that a projected execution is obtained by applying Π to each state in the execution sequence. This is almost the appropriate definition, but overlooks the situation in which consecutive states of the detailed execution project to the same state—a possibility we certainly want to allow, so that projections can reduce the number of steps in an execution as well as summarize state.

- Let $\langle s_i \rangle_i$ be a sequence of states. We define $\Pi(\langle s_i \rangle_i)$ to be the sequence $\langle \Pi(s_{i_j}) \rangle_j$ where $\langle i_j \rangle_j$ is defined by:

$$i_1 = 1 \text{ and } i_{j+1} = \text{ the smallest } i > i_j \text{ such that } \Pi(s_i) \neq \Pi(s_{i_j})$$

This completes the set of definitions we need for the following result:

Result 6 Let G be an activity graph, and $\Pi(G)$ a projection. Then Π projects any execution in G into an execution in $\Pi(G)$.

Proof Let $\langle s_i \rangle_i$ be an execution in G, and $\langle s_{i_j} \rangle_j$ its projection. We must prove that for $j > 1$, $\Pi(s_{i_j})$ is a subsequent state to $\Pi(s_{i_{j-1}})$. In G, we know that s_{i_j} is a subsequent state to $s_{i_{j-1}}$, that the state change was localized at a node, and caused either by τ or by o. Let s_u be the state of u at $s_{i_{j-1}}$. Then if the change was caused by τ:

$$\begin{aligned} \Pi_u(\tau_u(s_u, \sigma_{u/a,a}(s_{u/a}))) &\subseteq \tau_{\Pi(a),a}(\Pi_u(s_u), \Pi_{u/a,a}(\sigma_{u/a,a}(s_{u/a}))) \\ &= \tau_{\Pi(a),a}(\Pi_u(s_u), \sigma_{\Pi(u/a),a}(\Pi_{u/a,a}(s_{u/a}))) \end{aligned}$$

By $\tau_{\Pi(u)}$, we of course mean $\tau_{\Pi_u(d_u)}$, where d_u is the activity description that G places on node u. Thus, $\Pi(s_{i_j})$ is subsequent to $\Pi(s_{i_{j-1}})$. The case for o is similar.

The other requirement for an execution is that the values along an arc satisfy the protocol. This is true for the sequence of states seen in $\Pi(G)$ because it holds for these in G, and by the consistency requirement on the projections of values.

□

Whatever can be ascertained about executions in the view can be understood as a result about the detailed execution. Even when using the view as an inspection device, the condition that it is a projection means that the viewer can understand what is and what might happen next in a self-contained way—specifically, in terms of σ, τ, and o in the projections.

2.5 Summary

We have defined a graph-based scheme for the computational modeling and monitoring of activities. The basic building block is an *activity-description* (*activity description*), which describes "local" activity. An activity description governs how its instances can be connected to other instances of activity descriptions by specifying a *protocol* for each neighbor. It also governs the evolution of the state of an instance of the activity description: the parts of the state that are revealed to neighbors, the rules for changing that part of a state, and the rule for changing the part of the state not seen by any neighbor.

Structurally, an *activity graph* is a graph whose nodes are labeled with activity descriptions. Arcs between nodes receive two protocols, one from the activity description on the node at each end. There is a conformance rule requiring that the two protocols be the same, up to a symmetry operation. Activity graphs are allowed to have dangling arcs, each of which receives only one protocol. The set of protocols given by dangling arcs plays the same role as the set of protocols specified by an activity description.

We defined the notion of an execution of activity graphs. This consists of execution steps at each of the nodes, governed by the activity description on a node. At any step, a node may change its state independently of its neighbors, or it may change its state in a way that depends upon the part of a single neighbor's state made visible to it. The execution steps are required to be serializable: adjacent nodes may race to perform an interaction along an arc, but one or the other will go first. The sequence of values exposed along an arc must obey the protocol on the arc.

It is important that the definition of execution not require any central component in an implementation. We wish to support applications in which the nodes represent loosely dependent activities, and in which the activities may be proceeding at distant sites. Thus the formalism makes quite explicit exactly where the communication happens at each stage, and it requires only point-to-point communication, so there is no need for a global communication or locking mechanism.

Much of the discussion here has been toward the development of a calculus of operations on activity graphs. The first part of this discussion showed how to preserve the execution under a set of natural graph-theoretic transformations, while the second part left the graph structure invariant and showed how to characterize summarization of execution.

The reader may be concerned that activity graphs as a formalism do not provide an immediate basis for the construction of activity coordination programs by novices. In fact, there is no claim to the contrary, and as we stated at the outset, our goal here is to provide the intellectual basis for an activity coordination system. This basis must meet other criteria: it must provide a coherent "mentality" for an eventual system, by providing concepts that are applicable in its seemingly disparate parts. In chapter 5, we shall see how the somewhat theoretical ideas developed here relate to real world aspects of an activity coordination system, and in chapter 3 we develop several higher level activity descriptions and operations on activity graphs.

3 Some Common Idioms

The activity graph formalism was deliberately designed to favor formal simplicity and composability over the inclusion of many "features" at a fundamental level. The rationale is that if the basic design is clean and powerful, the desired features can be programmed and encapsulated. In the sections below, we consider common patterns of graph manipulation that can be done using the fundamental operations described in section 2.3. The aim here is not to be complete, but to demonstrate that a high-level system can indeed be built on this foundation.

3.1 Deleting Arcs and Nodes

Unlike the basic graph operations previously discussed, equivalence-preserving transformations that delete graph structure require semantic conditions. For arcs, a sufficient condition is that activity descriptions on the nodes at each end of the arc behave independently of the value seen along the arc. To be more precise, we say that d *ignores arc* $i_0 \stackrel{\text{def}}{\Leftrightarrow} \tau_{d,i_0}$ is independent of its second argument, i.e., there is a function $\theta_d : \mathcal{S}_d \rightarrow 2^{\mathcal{S}_d}$ such that:

- $\tau_{d,i_0}(s, v) = \theta_d(s)$, for all $s \in S_d$.

The fundamental transformation is on activity descriptions:

Let the activity description on the left be d with signature $(n_i : p_i)_i$ and assume d ignores i_0; for convenience, that on the right will be d' with signature $(n_i : p_i)_{i \neq i_0}$ and functions:

- $\sigma_{d',i}$ and $\tau_{d',i}$ are the same as $\sigma_{d,i}$ and $\tau_{d,i}$, respectively, for $i \neq i_0$.
- $o_{d'}(s) \stackrel{\text{def}}{=} o_d(s) \cup \theta_d(s)$

Suppose that a graph has nodes u_1 and u_2 joined by an arc a_0, and that each of u_1 and u_2 has an activity description that ignores a_0. The operations to delete the arc may be expressed as follows:

- Pinch the nodes at each end of the arc, and shrink a_0 (now a loop), obtaining the following functions for now node u_0:
  - $\sigma_{0,a}(\langle s_1, s_2 \rangle)$ for $a \neq a_0$ carries over from $\sigma_{i,a}(s_i)$, where a was incident upon u_i.
  - Similarly for $\tau_{0,a}(\langle s_1, s_2 \rangle, v)$.
  - $$o(\langle s_1, s_2 \rangle) = o_1(s_1) \cup o_2(s_2) \cup \tau_{1,a_0}(s_1, \sigma_{2,a_0}(s_2)) \cup \tau_{2,a_0}(s_2, \sigma_{1,a_0}(s_1)) = o_1(s_1) \cup \theta_1(s_1) \cup o_2(s_2) \cup \theta_2(s_2)$$

The first equality comes from shrinking $a_{0,i}$ and the second, from the assumption that u_i ignores a_0, for $i = 1$ and 2.

- Unpinch u_0 to obtain u'_1 and u'_2 with activity descriptions $d'_j, j = 1$ and 2, where:
  - $\sigma_{d'_j,a}$ and $\tau_{d'_j,a}$ for $a \neq a_0$ are the same as $\sigma_{d_j,a}$ and $\tau_{d_j,a}$, respectively.
  - $o_{d'_j}(s) = o_{d_j}(s) \cup \theta_{d_j}(s)$

  Thus u'_j has the above-described transformation of d_j.

The results for pinching and shrinking guarantee that the graph with the deleted arc has executions in 1-1 correspondence with the original graph.

Why would anyone write an activity graph with a deletable arc? One probably would not, at least not directly. The real utility of this transformation is in its combination with projection—even though an activity description does not ignore i_0, a projection might. Thus projection not only simplifies states and shortens execution sequences, it can also be viewed as simplifying the pattern of coordination. This is a formal property that corresponds to real-world experience: at a gross level, certain activities may be described as independent, while if one takes a closer look, it becomes clear that dependencies do exist.

For deleting a node, the condition is that its activity description has the following property:

d is *boring* $\stackrel{\text{def}}{\Leftrightarrow}$ its signature is empty and $o_d(s) = \emptyset$ for all s.

Let a node u_1 have a boring activity description. Then u_1 can be deleted as follows:

- Pinch u_1 together with any other node u_2, obtaining the following functions:
  - $\sigma_{d_0}(\langle s_1, s_2 \rangle) = \sigma_{d_2,i}(s_2)$
  - $\tau_{d_0}(\langle s_1, s_2 \rangle) = \{s_1\} \times \tau_{d_2,i}(s_2, v)$
  - $o_{d_0}(\langle s_1, s_2 \rangle) = o_{d_1}(s_1) \times \{s_2\} \cup \{s_1\} \times o_{d_2}(s_2) = \{s_1\} \times o_{d_2}(s_2)$

  The first two items use the fact that u_1 has no incident arcs, and the last, the fact that $o_{d_1}(s) = \emptyset$.

- Because d_1 enters into no interaction that would change its state, and because $o_{d_1}(s) = \emptyset$, the initial state of u_1 remains forever unchanged, i.e., s_1 in the above equations is a constant. Thus there is a 1-1 correspondence of states between u_2 and u_0, given by $s_2 \leftrightarrow \langle s_1, s_2 \rangle$. Hence, we can map the state on u_0 back to what it would have been on u_2, and revert to the original σ_{d_2}, τ_{d_2}, and o_{d_2}, and have a 1-1 correspondence in execution.

In effect, no trace of u_1 remains. Of course, an implementation would delete u_1 directly, and not literally go through the steps that justify doing so.

As with ignoring an arc, one is not going to write boring activity descriptions on purpose; their utility arises in connection with projection and other transformations. In the next section we give an example that combines projection and deletion of both arcs and nodes.

3.2 Subgraph Extraction

The goal of this section is to "understand" a subgraph G_0 of a given graph G on its own terms, where by "on its own terms" we mean that we wish to leave G_0 itself untouched, and to view interactions along boundary arcs of G_0 as contributing to the non-determinism of execution in G_0. More formally, our strategy is to define a projection on G which is the identity on nodes in G_0. The question then is, what happens to nodes outside G_0? We first consider the special case in which such a node has only one arc, the other end of which touches a node in G_0, and has $(n : p)$ as the signature for its activity description. In order to make the projection independent of the graph outside G_0, what we need is the *universal activity description for protocol p with name n*, denoted $d_{p,n}$. The definition is a bit technical, but the idea is simple: $d_{p,n}$ always responds to a sequence of interactions in a legal way, but is unpredictable up to the constraints imposed by p. The easy part is this:

- The signature for $d_{p,n}$ is $(n : p)$.
- The other constituents depend only upon p, and will be denoted $\mathcal{S}_p$, σ_p, τ_p, and o_p.
- $o_p(s) \stackrel{\text{def}}{=} \emptyset$ for all $s \in \mathcal{S}_p$.

The technicalities for the definitions of $\mathcal{S}_p, \sigma_p$, and τ_p are in Appendix A, where it is shown that there is a projection from d to $d_{p,n}$.

Generalizing the situation, suppose that a node u not in G_0 has activity description d with signature $(n_i : p_i)_i$. By "i touches G_0", we mean that the other end of the arc whose u-relative name is n_i touches a node in G_0. The projection for u takes d to an activity description that behaves on arc i like the universal activity description for p_i if i touches G_0, and which otherwise ignores arc i. Formally:

- $\Pi((n_i : p_i)_i) \stackrel{\text{def}}{=} (n_i : \left\{ \begin{array}{ll} p_i & \text{if } i \text{ touches } G_0 \\ \text{true} & \text{otherwise} \end{array} \right\})$
- $\mathcal{S}_{\Pi(d)} \stackrel{\text{def}}{=} \bigtimes_{i \text{ touches } G_0} \mathcal{S}_{d_i}$
- $\Pi(\sigma_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \sigma_{p_i}(s_i) & \text{if } i \text{ touches } G_0 \\ 0 & \text{(or any other arbitrary fixed value) otherwise} \end{array} \right.$
- $\Pi(\tau_{d,i})(\langle s_j \rangle_{j \text{ touches } G_0}, v) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \times_{j \text{ touches } G_0} \left\{ \begin{array}{ll} \tau_{p_i}(s_i, v) & \text{if } j = i \\ s_j & \text{otherwise} \end{array} \right\} & \text{if } i \text{ touches } G_0 \\ \emptyset & \text{otherwise} \end{array} \right.$
- $\Pi(o_d)(\langle s_j \rangle_{j \text{ touches } G_0}) \stackrel{\text{def}}{=} \emptyset$

We omit the proof that there is a projection from d to the activity description with these components.

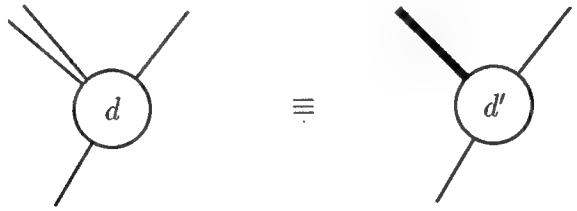
Now let us examine the result of this projection. We first observe that an arc between two nodes not in G_0 is ignored by both nodes, and hence may be deleted, by the previous section. After deleting all the arcs, any nodes that are not adjacent to a node in G_0 will be

boring, and thus may be deleted, again by the previous section. Finally, any remaining node not in G_0 will have arcs to one or more nodes in G_0. If the number of arcs is greater than 1, it may be unpinched[1] until all nodes not in G_0 have a single arc that touches a node in G_0. This is the last step of the construction—we now have a graph that embeds G_0 but is completely independent of the graph structure of G not in G_0.

The purpose of pursuing this is example is to offer evidence of the power of the pinching, shrinking and projection operations. Starting with an informal notion of wanting to "understand" a subgraph of a graph, the exercise of formalizing this in terms of the operations of sections 2.3 and 2.4 leads inevitably to the necessity of constructing an activity description that behaves as the most general participant in a protocol, which we called the universal activity description for a protocol. If we want merely to look at a subgraph of a graph, it would be overkill to invoke all this machinery. However, if we want to simulate or analyze a subgraph, or if we want to test and debug an activity protocol with a non-empty signature, an implemented version of $d_{p,n}$ is exactly what we need. So the seemingly technical device that we used to formalize extracting a subgraph from a graph turns out to have important connections to reality.

3.3 Collapsing Parallel Arcs

Section 2.3 showed how to condense any subgraph of a graph to a single node u_0 without loops. Consider a node u_1 outside the subgraph, which in the original graph had arcs to distinct nodes in the subgraph. In the transformed graph, the nodes u_0 and u_1 will be connected by several arcs. Arcs that are incident on the same pair of nodes are called *parallel*; the purpose of this section is to describe how parallel arcs can be collapsed to a single arc, and the activity descriptions on the nodes adjusted so that there is still a 1-1 correspondence in executions. The essence of the transformation is on an activity description:



The heavy arc on the right is obtained by collapsing the two arcs near each other on the left. Let the activity description on the left be d with signature $(n_i : p_i)_i$. For convenience, let n_0 be the name for the heavy arc. (Choose $n_0 \neq n_i, i > 2$.) The signature for the new activity description d' will be $(n_0 : p_0, n_3 : p_3, \ldots, n_k : p_k)$, where we have not yet defined p_0. Before doing so, we note that d' inherits the set of states from d, the definitions of $\sigma_{d,i}$ and $\tau_{d,i}$ for $i > 2$, and the definition of o_d.

The idea is that a value seen along the collapsed arc consists of the pair of values seen along the two uncollapsed arcs. There are two technical assumptions that we must make: the first is that $V \times V \subseteq V$, i.e., if $v_1, v_2 \in V$, then $\langle v_1, v_2 \rangle \in V$. The second has to do

[1]The pinching and shrinking transformations are equivalences, and may be applied in either direction; by "un-", we mean in the direction opposite to the one stated.

with a corresponding operation on protocols. Given $p_1, p_2 \in P$, we assume that there is a $p_1 \times p_2 \in P$ with the following action as a predicate:

- $(p_1 \times p_2)(\langle\langle f_i, v_i\rangle\rangle_i) \stackrel{\text{def}}{\Leftrightarrow}$ every v_i can be written in the form $\langle v_{i1}, v_{i2}\rangle$, and $p_j(\langle\langle f_i, v_{ij}\rangle\rangle_i)$ holds for $j = 1$ and 2.

We note that $(p_1 \times p_2)^T = p_1^T \times p_2^T$, i.e., they have the same action as predicates.

Naturally, we let $p_0 \stackrel{\text{def}}{=} p_1 \times p_2$, thus completing the signature for d', and to complete the definition of the entire activity description:

- $\sigma_{d',0}(s) \stackrel{\text{def}}{=} \langle \sigma_{d,1}(s), \sigma_{d,2}(s)\rangle$
- $\tau_{d',0}(s, \langle v_1, v_2\rangle) \stackrel{\text{def}}{=} \tau_{d,1}(s, v_1) \cup \tau_{d,2}(s, v_2)$
- $o_{d'}(s) \stackrel{\text{def}}{=} o_d(s)$

Result 7 Let u_1 and u_2 be nodes in a graph with activity descriptions d_1 and d_2, and suppose that a_1 and a_2 are parallel arcs adjoining them; for convenience, assume that a_i is the i^{th} arc on each node. Obtain a new graph by collapsing a_1 and a_2 to a single arc a_0, and by replacing the activity description on u_i by d_i', as outlined above. Then there is a 1-1 correspondence between the execution of the two graphs.

Proof We can view this as a sequence of elementary transformations:

- Pinch u_1 and u_2, then shrink a_1 and a_2, now loops, obtaining the following functions for the new node u_0.
  - $\sigma_{0,a}(\langle s_1, s_2\rangle)$ for $a \neq a_1, a_2$ carries over from $\sigma_{i,a}(s)$.
  - Similarly for $\tau_{0,a}(\langle s_1, s_2\rangle, v)$.
  - $$\begin{aligned} o_0(\langle s_1, s_2\rangle) &= o_1(s_1) \times \{s_2\} \cup \{s_1\} \times o_2(s_2) \\ &\quad \cup \tau_{1,a_1}(s_1, \sigma_{2,a_1}(s_2)) \times \{s_2\} \cup \{s_1\} \times \tau_{2,a_1}(s_2, \sigma_{1,a_1}(s_1)) \\ &\quad \cup \tau_{1,a_2}(s_1, \sigma_{2,a_2}(s_2)) \times \{s_2\} \cup \{s_1\} \times \tau_{2,a_2}(s_2, \sigma_{1,a_2}(s_1)) \\ &= o_1(s_1) \times \{s_2\} \cup \{s_1\} \times o_2(s_2) \\ &\quad \cup \tau_{d,0}(s_1, \sigma_{d',a_0}(\langle s_1, s_2\rangle)) \times \{s_2\} \\ &\quad \cup \{s_1\} \times \tau_{d',a_0}(s_2, \sigma_{d',a_0}(\langle s_1, s_2\rangle)) \end{aligned}$$
- Unshrink a new arc a_0 and then unpinch to get nodes u_1', u_2' with:
  - $\sigma_{u_i',a}(s_1) = \begin{cases} \sigma_{i,a}(s_i) & \text{if } a \neq a_1, a_2 \\ \sigma_{d',a_0}(s_i) & \text{if } a = a_0 \end{cases}$
  - Similarly for $\tau_{u_i',a}$
  - $o_{u_i'}(s_i) = o(s_i)$

The protocol $p_1 \times p_2$ has clearly been constructed to be valid for the communication on a_0. □

Thus, parallel arcs can always be collapsed, assuming that $V \times V \subseteq V$ and that P is closed under the product operations discussed above.

We state without proof several relationships that collapsing arcs has with other operations. First, shrinking and collapsing commute:

Result 8 Let i_1, i_1', i_2, i_2' be distinct indices of the signature of activity description d, and form d' by collapsing i_1 and i_2, and i_1' and i_2', calling the result indices i_0 and i_0'. Then:

$$d - \{[i_1, i_1'], [i_2, i_2']\} \cong d' - [i_0, i_0']$$

□

Result 9 Let i_1, i_2, and i_3 be distinct indices of the signature of activity description d, form $d_{\langle 1,2\rangle,3}$ by collapsing i_1 and i_2, and the resulting dangling arc with i_3, and form $d_{1,\langle 2,3\rangle}$ by collapsing i_2 and i_3, and then collapsing i_1 with the resulting dangling arc. Then:

$$d_{\langle 1,2\rangle,3} \cong d_{1,\langle 2,3\rangle}$$

□

At the beginning of this section, we remarked that collapsing parallel arcs is useful in tidying up a graph which has had a subgraph condensed to a node. In the next section, we shall see another use for this graph transformation.

3.4 Products of Isomorphic Graphs

A common pattern of activity is the assignment of essentially the same task to several persons, for example, all members of a committee are to receive a report and submit a review by a certain date. It is quite natural to describe what a committee member does from the point of view of one member; from the point of view of the person who assigns the task to the committee and who collects the reviews, it is natural to look at one graph that summarizes the behavior of all the members of the committee.

The notion of projected execution (section 2.4) supplies precisely the right technique for obtaining from the single graph, the view that is specific to a particular committee member. What is needed is a way to obtain the single activity graph that captures the behavior of a set of participants from a graph that specifies the behavior of a single participant, and does so in such a way that it is possible to project the appropriate view for each particular participant.

As the title of this section suggests, we will formalize the notion of "essentially the same task" to mean that the activity descriptions are based on isomorphic graphs. We do not require that the activity descriptions on corresponding nodes be in any way related, but we will assume that arcs which are paired by the isomorphism have the same names—this is not a real restriction, since renaming is always possible. The details:

- Let G_1 and G_2 be isomorphic activity graphs. This product, denoted $G_1 \times G_2$, is constructed as follows:

- Pinch each pair of nodes that is paired by the isomorphism.
- For each pair of dangling arcs paired by the isomorphism, transform the activity description on the node as described in section 3.3.
- For each pair of non-dangling arcs paired by the isomorphism, collapse the arcs as described later in the same section.
- Let the signature of G_i be $(n_j : p_{ij})_j$; the signature of $G_1 \times G_2$ is evidently given by $(n_j : p_{1j} \times p_{2j})_j$.

We state without proof several desirable properties of this construction. The first says that the product is associative and commutative, so that we can write $G_1 \times G_2 \times G_3$ without ambiguity, and similarly $\times_{i \in I} G_i$, where I is an index set. Further, there is an identity element, so the latter makes sense even if $I = \emptyset$.

Result 10 Let G_1, G_2, G_3 all have isomorphic graph structure. Define 1_G to have the same graph structure, where the protocol on a vertex is the boring protocol with the appropriate signature. Then:

$$G_1 \times G_2 \cong G_2 \times G_1 \qquad G_1 \times (G_2 \times G_3) \cong (G_1 \times G_2) \times G_3 \qquad G_i \times 1_G \cong G_i$$

□

Product and projection work in the way one expects:

Result 11 Let G_1, G_2 be activity graphs such that $G_1 \times G_2$ is defined. Let:

- $\Pi_i(n_j : p_{1j} \times p_{2j})_j \stackrel{\text{def}}{=} (n_j : p_{ij})_i$
- $\Pi_i(d_1 \times d_2) \stackrel{\text{def}}{=} d_i$
- $\Pi_i(\langle s_1, s_2 \rangle) \stackrel{\text{def}}{=} s_i$
- $\Pi_i(\langle v_1, v_2 \rangle) \stackrel{\text{def}}{=} v_i$

Then $\Pi_i(G_1 \times G_2) \cong G_i$, for $i = 1$ and 2.

□

Finally, there is a natural connection between graphical collapse and products of protocols and graphs.

Result 12 Let G_1, G_2 be as above, and for any G, let d_G be the interaction protocol obtained by collapsing G to a single vertex. Then:

$$d_{G_1 \times G_2} \cong d_{G_1} \times d_{G_2}$$

□

4 User's View

We have this far discussed abstract properties of activity descriptions. A usable system must of course have concrete, implemented activity descriptions. These ultimately rest on a set of *primitive activity descriptions*, each of which supplies, via parameterization, a family of activity descriptions with related semantics. In discussing the formalism, activity descriptions were represented by labels inside round nodes; in the implementation, a user sees a distinctive node shape representing a primitive activity description and node contents indicating the parameter; similarly, the name on the end of an arc in the formalism is, in what the user sees, implicit in where the arc is connected to the node. It turns out that there is nearly a 1-1 correspondence between the node types that a user sees and the primitive parameterized activity descriptions. (When there is this correspondence, for brevity's sake we will use the term "node type" to refer to both, and note the exceptions.)

The most important way of extending the system is by the addition of a new primitive activity descriptions, and most subsections we discuss the currently implemented set. The first subsection discusses the protocols used by this set, and the last two subsections discuss composite activity descriptions and notation for distributing activity descriptions.

4.1 Protocols

There are only two essentially different kinds of protocols currently used. One kind describes arcs that may be viewed as carrying a value from one node to another. These protocols come in the complementary pairs `send`(t) and `receive`(t), where t is the type of values (not to be confused with node type) that move on the arc. Arcs with these protocols are drawn as directed arcs, where the barb is at the end that is formally labeled with `receive`(t). The t is implicit, so no protocol label actually appears. The intuitive image is that the node at the `send`(t) end of the arc (away from the barb) places a value on the arc, thereby *enabling* it. Once an arc is enabled, the node "sending" the value cannot retract it and cannot place another value on the arc until receipt of the value is *acknowledged* by the node at the `receive`(t) end. After seeing the acknowledgment, the sending node *disables* its arc and, upon seeing this, the receiving node indicates that it is *ready*, and the cycle can begin again. Thus, the values exposed at the send end of an arc alternate between v and "disable", and the values exposed at the receive end of the arc alternate between "ready" and "acknowledge".

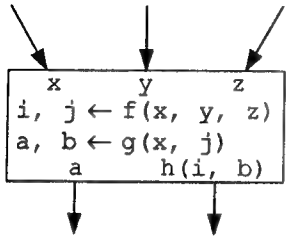
To relate this more closely to the formalism, the values exposed at the send end of an arc alternate between the enabling value $\langle 1, v\rangle$ (where v is the value being sent) and the disabling value $\langle 0\rangle$. The values exposed at the receive end of the arc alternate between the ready value 0 and the acknowledge value 1. The technical meaning of the `send`(t)/`receive`(t) protocol pair is a predicate on $(\{0,1\} \times V)^*$ which is true for sequences that alternate exposing values at ends of the arc, and at each end alternate as described above. In other words, these sequences have the form (where the `send` end of the arc is 0 and the `receive` end is 1):

$$\ldots, \langle 0, \langle 1, v_i\rangle\rangle, \langle 1, 1\rangle\langle 0, \langle 0\rangle\rangle, \langle 1, 0\rangle, \langle 0, \langle 1, v_{i+1}\rangle\rangle, \ldots$$

The other kind of protocol, called `contend`, is self-complementary. Arcs with this protocol are drawn undirected. They denote a contention between two nodes and are used in cases

where communication may be initiated at either end.

4.2 Nodes with Only `send/receive` Arcs



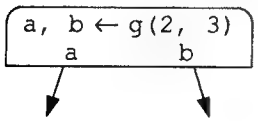
In an activity graph, a node of type *procedure-based* is represented as a rectangle with some text, as at the left. The semantics are that values arrive on incoming arcs in waves; new values are computed from these input values, and values are sent along the output arcs, as indicated by the expressions that label them. The parameter is a tuple consisting of the number of input arcs, the number of output arcs, the procedure (represented as an expression tree), and for each output arc, an indication of whether it is initially enabled, and if it is, the value being sent.[1]

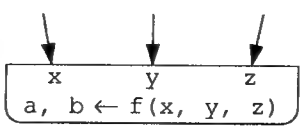
In many situations the only significance of the value on the arc is the fact that it is enabled. Such arcs can be viewed as transmitting values of type **none**, for which there is only one value. A value of type **none** can be transmitted from a procedure-based node simply by leaving its label blank. Similarly, a blank label on an input arc means that, regardless of type, the value on the incoming arc is ignored; however, the arc's role in synchronization remains. Thus, a procedure-based node with blank labels on all input arcs, blank labels on all output arcs, and no intervening expression lines acts as a synchronizer—it waits for an input wave, then sends an output wave, waits for another input wave, and so on. A node with type *synchronizer* is a graphical abbreviation for just such a procedure-based node.

(It uses the same primitive activity description as a procedure-based node.) Graphically, the node is a "." (as if it were an extremely condensed rectangle) so it can't really be seen as such in a graph. The barbs on input arcs to a synchronizer node are omitted, because their presence would be too messy graphically. An example is in the inset, where the two curved arcs converge on the straight arc.

When connecting arcs to a synchronizer node, it is not necessary to specify whether it is a **send** or **receive** arc. Arcs that connect to a procedure-based node (or other similar nodes that we will discuss below) have their orientation fixed by where they connect to that node. The only even slightly ambiguous arcs go from synchronizer node to synchronizer node. Instantiation uses a simple graph-based algorithm to determine whether there exists a unique orientation for such arcs subject to the constraint that every synchronizer node has at least one input and at least one output arc. If so, it uses that orientation; if not, it reports an error to the user.

Closely related to the procedure-based node types are *initialize* and *terminate*. An initialize node differs from a procedure-based node with no inputs in that the former produces only a single wave of outputs, while the latter can produce any number of waves. A terminate node differs from a procedure-based node with no outputs in that the former retains the values in its state and blocks further input, while the latter clears out values and will accept any number of input waves.



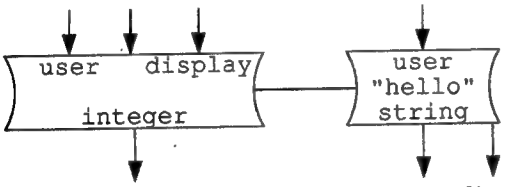


[1] In most cases, an output arc is not initially enabled. However, a cycle of **send/receive** arcs through procedure-based nodes must have an enabled arc, or execution is deadlocked.

Another node type having only `send/receive` arcs is *merge*. A merge node is parameterized by the number of input and the number of output arcs. It takes a value arriving on any input arc as soon as it arrives and sends it along all its output arcs. It does not require input to come in waves and will take arbitrarily many consecutive values from a single arc, but it will not take two consecutive values from a single arc, unless no values arrive on other arcs.

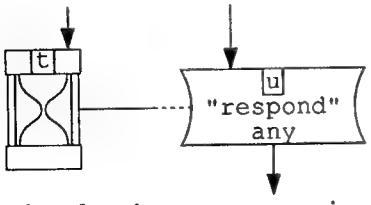
4.3 Nodes with `contend` Arcs

A *user-interaction* node is responsible for displaying values to users and accepting user input. Like a procedure-based node, a user-interaction node has input arcs and output arcs, responds to waves of inputs, and produces waves of output. The crucial difference is the period between the receipt of a wave of inputs and sending a wave of outputs, during which time the node is said to be *enabled*. During this time there are three values of interest: the user, the value being displayed, and the type that the user is expected to input. These values may arrive on arcs or may be specified by expressions in the node. This example shows only two possibilities—in both, the user arrives on an arc and the type is specified by a (constant) expression (on an output arc), while on the left the display arrives on an arc and on the right it is specified by the expression `"hello"`. When a node is enabled, it notifies the user, who responds with a value of the desired type, which is then transmitted along the arc labeled by the type. Note the middle incoming arc on the left node and the right outgoing arc on the right node. These are called *synchronization arcs*, because they transmit only `none` values, i.e., they behave as for a synchronizer node. Such arcs are distinguished by the fact that they connect to unlabeled places on the node.



A user-interaction node, unlike a procedure-based node, may have *exclusion arcs*, i.e., ones whose protocols are `contend` (represented in the above example by the horizontal arc connecting the nodes). The rule is that if both nodes are enabled, input will be accepted only from one—input at one node will disable the other node and, if there is a race, will resolve it. Note that widely separated users may each attempt an input, not knowing what the other is doing. If they each hit the "enter" key sufficiently close in time, one of their inputs will be ignored. In general, a user-interaction node may have exclusion arcs connected to any number of other nodes; in a race, the nodes decide that one and only one of the nodes successfully accepts the input, i.e., produces an output wave.

A *timer* node permits activity descriptions to interact with real time. Like a user-interaction node, a timer node alternates between being enabled and disabled, but when it is enabled, there is only one value of interest: the time at which it will "go off", i.e., send a wave of outputs. (On a timer node, all output arcs are synchronization arcs.) As with the values for a user-interaction node, the time can come in on an arc or be specified by an expression. In the example, the time expression is `t` and the user expression is `u`; both `t` and `u` are presumably parameters in the activity description of which this might be a part. The boxes around "t" and "u"

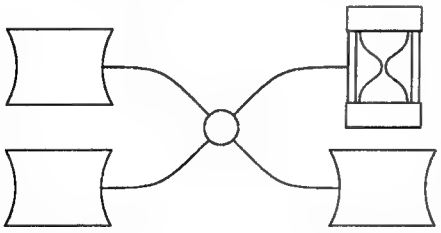


indicate that these are expressions, to which no arc can be attached, as opposed to the unboxed parameter "`user`" in the previous example, which indicates that the user value comes in along the arc attached to this part of the node.

Like a user-interaction node, a timer node can also have exclusion arcs. If the arc shown had continued undashed all the way to the user-interaction node, it would specify the symmetric mutual exclusion of the timer going off and the user responding to the user-interaction node. From the point of view of the user-interaction node, the dashed arc is called an *inbound-exclusion arc*, because the timer node can disable the user-interaction node even though the user-interaction node cannot disable the timer node. As an example that makes good use of inbound-exclusion arcs, consider several user-interaction nodes that have `contend` arcs to a timer node that are inbound-exclusion arcs from the point of view of the user-interaction nodes. This allows any subset of them to accept inputs without disabling the timer; when time runs out, the timer will disable the rest. Timer nodes may also have inbound-exclusion arcs.

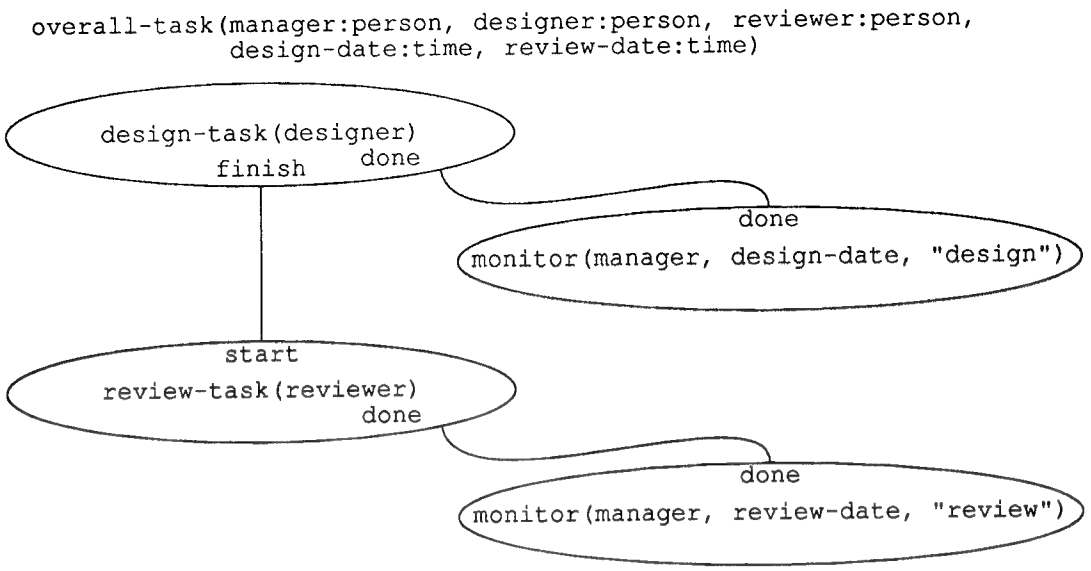
The user-interaction and timer nodes are semantically quite similar, differing only in that when enabled, one responds to the passage of time and the other to user input. (Both of these correspond to out-of-graph state changes in the activity graph formalism.) The similarity is captured in the implementation, in that they both use the same primitive activity description, *pbx*, meaning "procedure-based with exclusion". The parameterization is too complicated for this level of presentation, but the crucial aspect is that where a procedure-based node has an expression, a pbx node has three expressions, one to enable the node, one to handle normal arrival of the disabling event (user input or passage of time), and one to clean up after begin disabled by a contention arc. Thus, the addition of similar, seemingly complicated types of node, is in fact no more difficult then writing a procedure for a procedure-based node.

The only remaining node type with `contend` arcs has only this kind of arc. Its role is simply to pass along exclusion requests and is thus called an *exclusion* node. The graph in the inset is equivalent to one in which the exclusion (circle) node has been removed and the remaining nodes are connected pairwise by `contend` arcs. If one of the original arcs were an inbound-exclusion arc to a user-interaction or timer node, then all of the pairwise `contend` arcs would be inbound-exclusion arcs to that node.



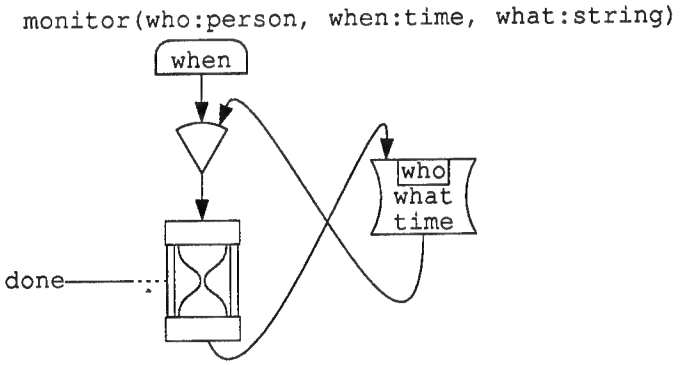
4.4 Composite Activity Descriptions

A *condensation* node, denoted by an ellipse, does not correspond to a primitive activity description; rather, it allows one activity description to refer to another activity description that has been specified by a graph, in the spirit of section 2.3. The following example illustrates the definition of a composite activity description (`overall-task`) that refers to several composite activity descriptions (`design-task`, `reviewer-task`, and `monitor`) defined elsewhere.



Because **overall-task** has no dangling arcs, it can be executed, at which time the parameters (**manager**, **designer**, etc.) must be specified. The activity descriptions mentioned in the condensation nodes all have dangling arcs, which are connected as indicated.

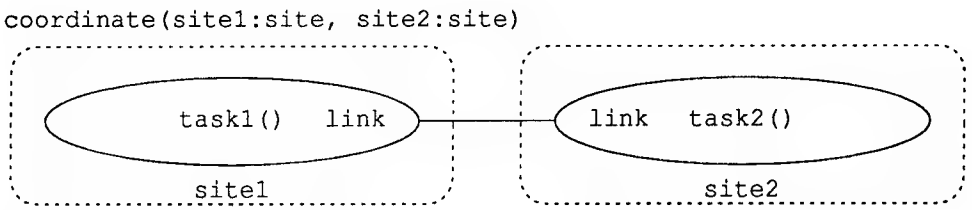
As a further example, we provide a plausible implementation of the **monitor** activity. The idea is that a timer is loaded with the initial deadline. If this timer goes off, it does not affect the monitored task (the timer has an inbound-exclusion arc), it merely alerts the manager (through the user-interaction node), at which point, a new deadline may be specified.



The dangling arc with label **done** corresponds to the same label in the elliptical condensation node.

4.5 Site Regions

This section is about multisite instantiation, not node types. One of the arguments to instantiation is the default site. All nodes are instantiated at this site, unless the activity description indicates otherwise. This is done by enclosing the subgraph to be instantiated at another site within a *site region*, an example of which is:



In this example, none of the graph actually runs at the default site.

5 Extender's View

An extender must necessarily have a deeper view of a system than the ordinary user. The first two sections below describe two aspects of the implementation that an extender must be aware of: the "long execution" facility within which nodes run, and how the implementation achieves concurrency in the presence of serializability semantics. The remaining sections discuss how the mathematical functions σ, τ, and o specify the functions that must actually be implemented.

5.1 Very Long Execution

Real-world activities continue for months or years, and consequently so must the instantiations that track them. For this purpose, the Activity Coordination System uses a "very long execution" facility [Kar87]. This facility provides *activations*, an analogue to operating system processes, but which survive machine crashes. Activations can send one another messages, even when they are at different sites on a wide-area net. Messages can also arrive from other sources, such as a timer or a process interacting with a user. When an activation is waiting for messages to arrive, or *asleep* for short, it exists solely in persistent memory. When a message arrives, an operating system process is spawned. This process reads in the state of the activation that was stored as it went to sleep and then begins receiving messages, starting with the one that woke it up. When no messages have arrived for a sufficiently long interval (e.g., a minute), the activation stores its new state and goes back to sleep.

5.2 Serializability and Concurrency

We have seen that there are strong mathematical reasons, based on practical system considerations, that a node be able to change its exposed value on only one arc at a time, and that it can do so with a secure knowledge of the value exposed at the other end of the arc. This mathematical model suggests that communication along an arc is instantaneous, while a goal of the implementation is to provide coordination over wide-area networks, possibly using email. The slowness of real-world communication gives rise to two problems—how does the system resolve races between two nodes, each of which wants to change the exposed value on a shared arc, and how can the slow communication on several arcs be overlapped?

We first describe the algorithm to resolve races. Consider two nodes connected by an arc yet separated by a time-consuming communication path. The graph instantiation mechanism obtains from the activity description in each node the initial value exposed on each arc and sends both this value and its address—network address, instantiation id, node within the instantiation, and arc on the node—to the other end of the arc. Initially, then, each node has a local copy of the value exposed on the other end of the arc, or at least, the most recently received such value; this invariant is maintained throughout execution, as we will see. The instantiation mechanism also tags one end of the arc with 1 and the other with 0, for reasons that will become apparent.

Changing the exposed value on an arc is a two step process. A node first proposes a new value, based on its local copy of the value at the other end. If the proposal is confirmed, it

proceeds with the state change, and otherwise, not. To propose a new value, a node sends a "change" message, consisting of the new value and a random bit, and waits. When a node receives a change message on an arc, there are two cases of interest: either it is waiting for a response to a change message sent out over this arc, or not. In the latter case, the node simply sends an acknowledgment, and records the received value as the new local copy. The former case indicates a race—the change messages crossed in the mail, and to understand the algorithm, it is useful to visualize each node performing the steps jointly. Each node computes the exclusive-or of the random bit it sent, the random bit it received, and the tag on its end of the arc; one node gets 0 and the other gets 1, so each "knows" what will happen at the other. The node that gets 0 views this as denying its proposed change and the node that gets 1 views this as confirming the proposed change, and treats the change message as an acknowledgment, so that no further messages need to be sent. Thus, each successful change of an exposed value requires only two message, whether or not there is a race.

We now discuss the use of concurrency. The issue is the wait between sending a change message and receiving an acknowledgment. While this wait is unavoidable, it would be desirable to overlap the waits on different arcs as much as possible. The problem in doing so is the unpredictability of whether the proposed changes will be confirmed, and if so, the order in which to make state changes. For example, suppose a node in state s_0 concurrently proposes changes to arcs a_1 and a_2. Let us first consider the case in which both changes are confirmed. The weakest condition that will satisfy the serializability rule is that for some ordering of the arcs, i.e.:

- $\exists \langle i, j \rangle \in \{\langle 1, 2 \rangle, \langle 2, 1 \rangle\}$: Let s_i be the state that exposes the value proposed for a_i. Then the values proposed for a_j in states s_0 and s_i must be the same.

In generalizing this to n concurrent changes, there are $n!$ possible orders to consider, and consideration would have to wait until all concurrently proposed changes are confirmed or denied. Thus an implementation based on this observation is infeasible.

To circumvent the problem of batching concurrently proposed changes, let us suppose that we use the strategy that a state change takes effect as soon as the proposed value is confirmed. This makes the algorithm that handles messages and state changes simple, but requires a stronger condition on state changes if it is to be correct: change "$\exists$" to "$\forall$" in the above condition. In generalizing this to n concurrent changes, there are still $n!$ possible orders to consider, but these need to be considered only in proofs or run-time checks.

This brings us closer to the idea on which the implementation is based, but there is one more issue to consider. In the above description of the algorithm, we glossed over exactly what it meant to "proceed with the state change". Because state changes can occur in any order, the state seen when the interaction is confirmed may be different from what it was when the value was first proposed. It is thus preferable to describe the state that holds after confirmation not directly by a new state value, but indirectly by a *delta* that is used to obtain a new state from an old one. For example, a delta may say "increment the counter" rather than "the new value of the counter is 3". Technically, a proposed change is a triple consisting of an arc, the new exposed value for the arc, and a delta. At any time in a node's execution, there will a set of proposed changes, with no two triples having the same arc. The

following condition on deltas allows the system to change state as acknowledgments arrive, and to drop proposed changes that lose a race:

- Let $\{\langle a_i, v_i, \delta_i \rangle\}_{i \in I}$ be the set of proposed changes in state s. Then:
  - For any $i \in I$, the set of proposed changes in state $\delta_i(s)$ and the set of proposed changes for s differ at most on elements whose arc is a_i.
  - For all $i, j \in I, i \neq j$, δ_i and δ_j commute, i.e., $\delta_i \circ \delta_j = \delta_j \circ \delta_i$.

This condition is stronger than the one above, but in practice, there seems little loss in generality. Further, because the δ_i are simple data structures, it is easy to implement these tests to check the correctness of concurrent changes.

5.3 Instantiation

At the most fundamental level, a new type of primitive activity description is added to the system by implementing its *instantiation function.* This is a function of one argument, the parameter for the primitive activity description. (If an activity description has several conceptual argument, they are bundled into a single argument, for example, a tuple of values.) The results of an instantiation function are:

- The initial *private state* of the node.
- The initial *out-of-graph set* for the node (explained below).
- A vector of strings, the node-relative names of arcs.
- A vector of values, giving the initial exposed value on each arc.
- A vector of *tau-functions* (explained below).

The initial state of the node, the arc names, and the initial exposed values are all obvious enough, modulo the issue of the representation that allows them to be maintained in persistent form, which we do not discuss here. The interesting part is the out-of-graph set and the tau functions, which is where the connection between the formalism and the implementation must be made.

5.4 Out-of-graph Functions, Sets, and Messages

The private state and the out-of-graph set of a node together comprise what we called the state of a node in the formalism. The reason for splitting the state in this way is motivated by how out-of-graph state changes occur: they arise because of the arrival of *out-of-graph messages.* The generic (i.e., node-type independent) machinery does not care about the details of most of the state; this is the private state, i.e., private to the implementation of the node. However, the generic machinery must be able to test the formal condition $o(s) = \emptyset$ (part of the termination test) and it must know what to do when an out-of-graph message arrives. The out-of-graph set solves both of these problems, as we shall see.

An out-of-graph set consists of key/oog-function pairs (with distinct elements having distinct keys) and an out-of-graph message is a key/value pair. When an out-of-graph message arrives at a node, the generic machinery looks up the key of the message in the out-of-graph set. If it is not there, the message is ignored—the node is not vulnerable to this kind of out-of-graph state change. If it is there, then the oog-function is called; it has the following arguments and results:

- *oog-function: private-state, oog-set, value → delta-private-state, delta-oog-set*

We can be quite precise about the relationship between an implemented oog-function f and the function o of the formalism. First, let the private state and oog-set be p and q respectively; the set s of the formalism is effectively $\langle p, q \rangle$. Let δ_p, δ_q be the result of $f(p, q, v)$, so that the new set is given by $s' \stackrel{\text{def}}{=} \langle \delta_p(p), \delta_q(q) \rangle$. (The pair $\langle \delta_p(p), \delta_q(q) \rangle$ serves as the δ discussed in section 5.2, and must observe those constraints.) The specification for f is that $s' \in o(s)$. The non-determinism of the formalism, i.e., the fact that $o(s)$ is a set, is replaced by the choice of a specific state in the implementation, the choice aided by the value that arrived in the message, an aspect of reality not under the purview of the formalism.

Thus, the out-of-graph set not only says what to do with an out-of-graph message, but also provides the generic machinery with a way of testing whether $o(s) = \emptyset$: if the out-of-graph set is empty, all messages will be ignored, and no out-of-graph state change will occur.

5.5 Tau Functions

A *tau-function* is the implementation specified by a τ function of the formalism. It has the following arguments and results:

- *tau-function: private-state oog-set seen-value*
  → exposed-value delta-private-state delta-oog-set

As we discussed in section 5.2, a tau-function only proposes an interaction, or more precisely, the generic machinery initiates an interaction if and only if the "exposed value" result of a tau-function differs from the currently exposed value.

Let f be a tau-function, let p, q be the components of the implemented state, and let v be the seen value; let the results of $f(p, q, v)$ be v', δ_p and δ_q. The specification for f is:

- Let $s' \stackrel{\text{def}}{=} \langle \delta_p p, \delta_q q \rangle$. Then $s' \in \tau(\langle p, q \rangle, v)$ and $v' = \sigma(s')$

Note that σ in the formalism does not correspond to a separately implemented function, but is instead bundled into the implementation of a tau-function.

A tau-function is called under two circumstances, both of which are subject to the condition that there is no pending interaction on the arc to which the tau-function is attached (so that a new interaction will not be proposed if one is already pending). The first condition is when the value seen at the other end of the arc changes and the second is when the private state or out-of-graph set changes. (This can be caused by an oog-function or a successful interaction on some arc.)

A The Universal Activity Description for a Protocol

We show that for any protocol p, there exist $\mathcal{S}_p$, σ_p, τ_p, and o_p, which together with the signature $(n : p)$ comprise an activity description $d_{p,n}$ with the following property:

- For *any* activity description d with signature $(n : p)$, there is a projection from d to $d_{p,n}$.

Because of the generality of this result, in particular, the fact that p can be an arbitrary predicate, the proof we give is non-constructive—it does not say how to implement $d_{p,n}$. In practice, this is usually not difficult, but even if it is, the result says to keep trying, because the solution exists.

The almost right idea for an element of $\mathcal{S}_p$ is a sequence of pairs $\langle f_1, v_1\rangle, \ldots, \langle f_k, v_k\rangle$ for which p holds and which ends with $f_k = 0$. In this "state":

- $\sigma_p(\langle\langle f_i, v_i\rangle\rangle_{l=1}^{k})$ would be defined to be v_k.
- $\tau_p(\langle\langle f_i, v_i\rangle\rangle_{l=1}^{k})$ would be defined to be the set of all sequences $\langle\langle f_i, v_i\rangle\rangle_{i=1}^{k}$ where:
  - p holds for $\langle f'_1, v'_1\rangle, \ldots, \langle f'_l, v'_l\rangle$ and $f'_l = 0$ (otherwise this would not be a legitimate state).
  - The new sequence is an extension of the original, i.e., $l > k$ and $f'_i = f_i$ and $v'_i = v_i$ for $i = 1, \ldots, k$. Thus, we may drop the primes from f'_i and v'_i.
  - The new sequence does not skip any interactions at the node in question, i.e., $f_i = 1$ for $i = k+1, \ldots, l-1$. (If $l = k+1$, this is a vacuous condition.)
  - The new sequence is compatible with the value seen by τ, i.e., : $v = v_{i_0}$ where $i_o = \max\{i < l \mid f_i = 1\}$. (If $f_i = 0$ for all i, this is vacuous.)

The motivation behind this definition was stated earlier: τ_p responds to a sequence of interactions in a legal way, and it will respond to any legal sequence, i.e., is unpredictable up to the constraints imposed by p. The only reason that it is not quite right is that a state "remembers" too much: a sequence of states in an execution that we wish to project may repeat, but states in the above definition of τ always grow longer and thus never repeat. Hence we cannot obtain a projection.

This flaw can be remedied by using as states, i.e., elements of $\mathcal{S}_p$, not sequences of the above form, but rather, quotient sets of sequences of the above form, under the following equivalence relation:

- For $j = 1$ and 2, let $t_i \stackrel{\text{def}}{=} \langle\langle f_{ij}, v_{ij}\rangle\rangle_{l=1}^{k}$. We will assume that p holds for t_j and that $f_{k_j j} = 0$. We define $t_1 \sim t_2 \stackrel{\text{def}}{\Leftrightarrow}$
  - $v_{k_1,1} = v_{k_2,2}$ (both "states" expose the same value).
  - For any t_0, $p(t_1 \cdot t_0) \stackrel{\text{def}}{\Leftrightarrow} p(t_2 \cdot t_0)$ (where $\cdot$ means concatenation).

  Intuitively, p can't tell the difference between t_1 and t_2.

If we look back at the would-be definitions of σ_p and τ_p, we see that they depend only upon properties that are invariant under the above equivalence relation, and thus that σ_p and τ_p may be defined in the same way for equivalence classes.

Let d have signature $(n : p)$, and suppose that $s \in \mathcal{S}_d$. We must define the element of $\mathcal{S}_p$ to which s maps. In keeping with our non-constructive approach, look at all executions of d that can arrive at s, and extract from each such sequence s_j a sequence of pairs $\langle\langle f_{ij}, v_{ij}\rangle\rangle_i$, describing traffic along the arc. Since the definition of execution requires that p holds for the extracted sequence of pairs, and since the fact that s_j arrives at s implies that any two such sequences are equivalent using the above definition, the set of all such sequences of pairs is contained in an element of $\mathcal{S}_p$, which is then the desired $\Pi(s)$.

This leaves only the matter of defining o_p. Since an application of o_d does not appear in an extracted sequence of pairs, if $s' \in g(s)$, then $\Pi(s') = \Pi(s)$. Thus, out-of-graph changes in d are not noticed in $\mathcal{S}_p$, and:

- $o_p(s) \stackrel{\text{def}}{=} \emptyset$

This completes the definition of $d_{n,p}$.

B Bibliography

[Hoa84] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, England, 1984.

[Kar87] Michael Karr. Very long executions. Technical Report SOI-19-87, Software Options, Inc., July 1987.

[Kar90] Michael Karr. Transaction graphs: A sketch formalism for activity coordination. Technical Report RADC-TR-90-347, Rome Air Development Center, December 1990.

[Mor91] Robert T. Morris. Implementation of an activity coordination system. In *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference.* Rome Laboratory, September 1991.

[Pet] Advances in Petri Nets. Occasional volumes in Lecture Notes in Computer Science (Springer-Verlag).

*U.S. GOVERNMENT PRINTING OFFICE: 1995-610-126-50147

Rome Laboratory

Customer Satisfaction Survey

RL-TR-_____

Please complete this survey, and mail to RL/IMPS, 26 Electronic Pky, Griffiss AFB NY 13441-4514. Your assessment and feedback regarding this technical report will allow Rome Laboratory to have a vehicle to continuously improve our methods of research, publication, and customer satisfaction. Your assistance is greatly appreciated.
Thank You

_____

_____

Organization Name:_____(Optional)

Organization POC: _____(Optional)

Address:_____

1. On a scale of 1 to 5 how would you rate the technology developed under this research?

5-Extremely Useful 1-Not Useful/Wasteful

Rating_____

Please use the space below to comment on your rating. Please suggest improvements. Use the back of this sheet if necessary.

2. Do any specific areas of the report stand out as exceptional?

Yes___ No_____

If yes, please identify the area(s), and comment on what aspects make them "stand out."

3. Do any specific areas of the report stand out as inferior?

Yes___ No___

If yes, please identify the area(s), and comment on what aspects make them "stand out."

4. Please utilize the space below to comment on any other aspects of the report. Comments on both technical content and reporting format are desired.

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.